University of Alberta

MULTI-DISPATCH IN THE *Java* VIRTUAL MACHINE: DESIGN, IMPLEMENTATION, AND EVALUATION

by

Christopher John Dutchyn ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2002

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-69701-0

Canada

University of Alberta

Library Release Form

Name of Author: Christopher John Dutchyn

Title of Thesis: Multi-Dispatch in the *Java* Virtual Machine: Design, Implementation, and Evaluation

Degree: Master of Science

Year this Degree Granted: 2002

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Christopher Dutchyn

Christopher John Dutchyn
11032 135 Street NW
Edmonton, Alberta
Canada, T5M 1K5

Date: March 18, 2002

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Multi-Dispatch in the *Java* Virtual Machine: Design, Implementation, and Evaluation** submitted by Christopher John Dutchyn in partial fulfillment of the requirements for the degree of **Master of Science**.

Duane Szafron
Co-Supervisor

Paul Lu
Paul Lu
Co-Supervisor

Jonathan Schaeffer
Jonathan Schaeffer

Dennis Ward

Date: March 15, 2002

I find OOP technically unsound. It attempts to decompose the world in terms of interfaces that vary on a single type. To deal with the real problems you need multisorted algebras — families of interfaces that span multiple types.

Alexander A. Stepanov
Author — The C++ Standard Template Library

Abstract

Mainstream object-oriented programming languages, such as Java and C++, offer only a restricted form of dynamic polymorphic method selection, namely single-receiver dispatch. Several common programming patterns require developers to circumvent this limitation with additional, awkward, and type-specific code. This dissertation investigates the more general facility of multiple dispatch within the Java environment. We describe Java syntax that permits the programmer to enable multiple dispatch while retaining compatibility with existing single dispatch Java source and binary programs. We develop the semantics of multiple dispatch within the dynamic, reflective, and secure Java programming model. We describe an extension to the Java Virtual Machine that implements our programmer-directed multiple dispatch and retains compatibility with single dispatch Java programs and libraries. We demonstrate that our implementation imposes negligible time penalties on existing programs, yet offers the general multiple dispatch facility with performance equal to programmer-tuned double dispatch.

Preface

The single name displayed on the title page of any dissertation is typically a misleading fiction. This volume is no exception: it represents the time, energy, and ideas of many others, to whom I am grateful, and to whom I am forever indebted.

To Cadence: you never complained when I went to work, and always smiled when I came home.

To Cheryl, my wife and companion: without your support, patience, and understanding, I would not be here today.

To my supervisors, Duane Szafron and Paul Lu: progress is much slower than expected, even accounting for Cadence's birth. I am grateful for your patience, your support, your critical review, and your insight. I have learned much both directly and from your example.

To Jonathan Schaeffer, who chaired the graduate committee that looked beyond my past failures and admitted me to the programme after a lengthy departure, thank you for your faith.

To Candy Pang, Wade Holst, and Yuri Leontiev, I am grateful that you welcomed me into your research group, challenged me with tough questions, and accepted my "I don't know"s ... at least temporarily.

To Tom Harke: it's been a long journey from Antonelli's differential geometry class, but your sharp wit has been an always welcome foil.

Acknowledgements

Single Receiver Projections, an incremental multi-method resolution algorithm, is a marvel of economy. Wade Holst, Candy Pang, Yuri Leontiev, and Duane Szafron provided me with a sharp tool in my quest for high-performance Java multi-methods. I am grateful, and I hope that my enhancements to accommodate the peculiarities of Java, specifically dynamic class loading and `NULL` arguments, do not blemish the technique.

Near the end of the research, we realized that the proof of the system was in a large application. Steve Bromling bravely volunteered to re-cast the `Swing` and `AWT` libraries as a multi-method system. In doing so, he managed to tweak a few bugs in the multi-dispatch virtual machine, as well as strongly reinforce the argument for broad multi-method support: constructors, all method categories, arguments of array type, and co-variant return types. He also maintained a detailed log of the changes, enabling us to crudely measure the software engineering benefits the multi-methods offer. Without his efforts, a demonstration of the potential of this system would still be an unrealized dream.

Contents

1 **Introduction** 1
  1.1 Goal 3
  1.2 Terminology 3
  1.3 Organization 3
  1.4 Contributions 5

2 **Object-Oriented Languages and Dispatch** 6
  2.1 Uni-Dispatch 6
  2.2 Double Dispatch 8
    2.2.1 Binary Operations 8
    2.2.2 Drag and Drop 11
    2.2.3 Event-Driven Programming 11
    2.2.4 Publish-Subscribe 14
  2.3 Multi-Dispatch 15

3 **Dispatch in Java** 16
  3.1 Java Classfile format 16
  3.2 Static Multi-Dispatch in Javac 17
  3.3 Dynamic Uni-Dispatch in the JVM 24
    3.3.1 Resolution 25
    3.3.2 Lookup 27
    3.3.3 Lossless Bytecodes 31
    3.3.4 Invocation 33

4 **Multi-Dispatch Java — Design** 34
  4.1 Marking Methods For Multi-Dispatch 34
    4.1.1 Marker Interfaces 34
    4.1.2 Multi-Dispatchable Attributes 35
  4.2 Locating the Multi-Dispatcher 36
  4.3 The Multi-Dispatcher 37
    4.3.1 Multi-Dispatch Semantics in Java 38
    4.3.2 `Invoke-static` Multi-Dispatch 42
    4.3.3 `Invoke-nonvirtual` Multi-Dispatch 43
    4.3.4 `Invoke-super` Multi-Dispatch 44
    4.3.5 `Invoke-virtual` Multi-Dispatch 45
  4.4 Potential Errors 46
    4.4.1 Ambiguous Invocations 46
    4.4.2 Incompatible Return Types 47
    4.4.3 Accessibility 49
    4.4.4 Reporting Multi-Dispatch Errors 49

5 Multi-Dispatch Java — Implementation 51
5.1 Multi-Dispatch Placement . . . 51
5.1.1 Multi-Invoker . . . 51
5.1.2 Inline Dispatch . . . 52
5.2 Dispatch Techniques . . . 52
5.2.1 Reference Implementation: *MSA* . . . 53
5.2.2 Tuned SRP Dispatcher . . . 54

6 Multi-Dispatch Java - Evaluation 74
6.1 Compatibility . . . 76
6.2 Correctness . . . 79
6.3 Multi-Dispatch Performance . . . 79
6.3.1 Multi-Dispatch Versus Double Dispatch Performance . . . 80
6.3.2 Arity Effects . . . 85
6.3.3 `Swing` and `AWT` . . . 87
6.4 Summary . . . 88

7 Discussion 90
7.1 Related Work . . . 90
7.2 Future Work . . . 92
7.2.1 Just-In-Time Compilers . . . 92
7.2.2 Additional Optimizations . . . 93
7.2.3 Explicit `null` Parameters . . . 94
7.2.4 `super` On Non-Receiver Arguments . . . 95
7.2.5 Tuple Syntax . . . 96
7.2.6 Parametric Polymorphism . . . 96
7.2.7 Other Dispatchable Properties — Security . . . 97
7.2.8 Fully Multi-Dispatch Java . . . 98
7.3 Type-less Programming . . . 99
7.4 Closing Remarks . . . 100

Bibliography 101

A Performance Raw Results 104
A.1 Double Dispatch Raw Results . . . 104
A.2 Multi-Dispatch Raw Results . . . 106

B Dispatch Evaluation Code 107
B.1 The Driver Class . . . 107
B.2 Uni Dispatch . . . 108
B.3 Double Dispatch . . . 109
B.3.1 Visitor Pattern . . . 109
B.3.2 Typecases . . . 110
B.3.3 Inlined Typecases . . . 111
B.3.4 Typefields via Accessors . . . 112
B.3.5 Inlined Typefields via Accessors . . . 113
B.3.6 Direct Typefields . . . 114
B.3.7 Inlined Direct Typefields . . . 115
B.3.8 Event Dispatch Kernel . . . 116
B.4 Multi-Dispatch . . . 117
B.4.1 Simple Multi-Dispatch . . . 117
B.4.2 Fully Multi-Dispatch . . . 118

C Test Suite **119**
C.1 ByteCode Tests 119
C.1.1 `IVQ` 119
C.1.2 `IVQW` 120
C.1.3 `IVOW` 121
C.1.4 `ISQ` 122
C.1.5 `INVQ` 123
C.1.6 `INVSQ` 124
C.2 Integrated Tests 125
C.2.1 Multi-Dispatch Diamond 125
C.2.2 Array Types 126
C.2.3 `Invoke-virtual` Semantics 127
C.2.4 `Invoke-static` Semantics 131
C.2.5 `Invoke-special` Semantics 134

D Multi-Dispatch Placement Implementation **137**
D.1 MultiInvoker 137
D.2 Inlined Multi-Dispatch 138

E Multi-Dispatcher Implementation **139**
E.1 MSA Algorithm 139
E.2 Tuned SRP Algorithm 143

List of Figures

2.1 Dispatch Techniques in Standard Java 7
2.2 The Container Problem 8
2.3 Double vs. Multi-Dispatch in Java 9
2.4 Typecases / Binary Operations 10
2.5 Strategy Pattern / Drag and Drop 12
2.6 Multi-Dispatch Event-Driven Programming without Type Fields 13
2.7 Visitor Pattern / Publish Subscribe 14

3.1 A Simple Constant Pool for `ColorPoint` (Figure 2.1) 17
3.2 Ambiguous and Conflict Methods 19
3.3 Hierarchy of Primitive Types in the Java Language 19
3.4 Automatic Primitive Promotions 20
3.5 Loss of Precision 20
3.6 Arrays as Parameterized Types 21
3.7 Java Language Type Hierarchy 22
3.8 Java Virtual Machine Type Hierarchy 23
3.9 Polymorphic Call-sites — two views 24
3.10 Static Inheritance Through Resolution 26
3.11 A Constant Pool After Method Resolution 26
3.12 Simplified Object and Class Structure of instance of class `ColorPoint` 29
3.13 Array Element and Class Structure for array of `ColorPoint` 30
3.14 Summary of Quick Optimizations 32

4.1 Multi-Invoker Based Multi-Dispatch 37
4.2 Inlined Multi-Dispatch 38
4.3 Structure of the Multi-Dispatcher 39
4.4 Type Hierarchy Induces Method Type Lattice 41
4.5 Static and Virtual Methods Interfere 43
4.6 `Super` Invocations Skip Levels 44
4.7 Private Method Targeted By Multi-Dispatch 45
4.8 Ambiguous Dispatch and Conflict Method 47
4.9 Illegal Return Type Error 48

5.1 Behaviour Management 56
5.2 Behaviour `*.method(int,*)` 58
5.3 Type-Numbering Operations 62
5.4 Behaviour `*.method(int,*)` After Numbering `B` 63
5.5 Behaviour `*.method(int,*)` After Resolving All Methods 64
5.6 Ambiguous Dispatch in `Super.smethod(*)` 67
5.7 Unresolved Types 68
5.8 Dispatches for Figure 5.2 69
5.9 After Dispatching `(new Super()).method(1,new A())` 70
5.10 Dispatches for Figure 5.9 70

5.11 Dispatches for Figure 5.5 71
5.12 Find-First-Set Implementation 71
5.13 Extended SRP Dispatcher 72

6.1 Arity Effects Test Hierarchy 85
6.2 Impact of Arity on Dispatch Latency 86

7.1 Conflict Method Not Definable 95
7.2 `Tuple`-like Syntax 96
7.3 Array-based Parametric Polymorphism in Multi-Dispatch Java 97
7.4 Security and Multi-Dispatch 98

B.1 Double Dispatch Driver 107
B.2 Uni-Dispatch Basic Structure 108
B.3 Visitor Pattern 109
B.4 Typecases 110
B.5 Inlined Typecases 111
B.6 Typefields via Accessors 112
B.7 Inlined Direct Typefields 113
B.8 Direct Typefields 114
B.9 Inlined Direct Typefields 115
B.10 Event Dispatch Kernel 116
B.11 Simple Multi-Dispatch 117
B.12 Fully Multi-Dispatch 118

C.1 `IVQ` Test 119
C.2 `IVQW` Test 120
C.3 `IVOW` Test 121
C.4 `ISQ` Test 122
C.5 `INVQ` Test 123
C.6 `INVSQ` Test 124
C.7 Multi-Dispatch Diamond 125
C.8 Array Types Test 126
C.9 `invoke-virtual` Semantics Test (Part I) 127
C.10 `invoke-virtual` Semantics Test (Part II) 128
C.11 `invoke-virtual` Semantics Test (Part III) 129
C.12 `invoke-virtual` Semantics Test (Part IV) 130
C.13 `invoke-static` Semantics Test (Part I) 131
C.14 `invoke-static` Semantics Test (Part II) 132
C.15 `invoke-static` Semantics Test (Part III) 133
C.16 `invoke-special` Semantics Test (Part I) 134
C.17 `invoke-special` Semantics Test (Part II) 135
C.18 `invoke-special` Semantics Test (Part III) 136

D.1 MultiInvoker Implementation 137
D.2 Multi-Dispatch Assembler Interpreter Loop — Inline Test Macro 138

E.1 `SelectVirtualMultiMethod` for MSA Technique 139
E.2 `SelectStaticMultiMethod` for MSA Technique 140
E.3 `SelectSpecialMultiMethod` for MSA Technique 140
E.4 `SelectSuperMultiMethod` for MSA Technique 141
E.5 Inner Dispatcher for MSA Technique 142
E.6 SRP Dispatcher — Behaviour Structure 143
E.7 `MMDispatch` — The Outer Dispatchers 143
E.8 32-Bit Implementation of Inner Dispatcher 144

E.9 Resolver 144
E.10 `Select-*-MultiMethod` Routines 145

List of Tables

6.1 Multi-Dispatch Implementations 75
6.2 Uni-Dispatch Implementations 75
6.3 Summary of Results Tables 76
6.4 Compatibility Testing and Performance 77
6.5 Uni-Dispatch Compatibility Testing and Performance 78
6.6 Visitor Pattern Compatibility Testing and Performance 79
6.7 Multi-Dispatch Performance 82
6.8 Double vs. Multi-Dispatch — part I: MSA 83
6.9 Double vs. Multi-Dispatch — part II: SRP-L 84
6.10 Double vs. Multi-Dispatch — part III: SRP-C 84
6.11 Arity Effects Performance 86
6.12 `Swing` Application Method Invocations 87
6.13 `Swing` Application Execution Time 88

A.1 Double Dispatch Performance — part I: Event Kernel and Visitor Pattern . . 104
A.2 Double Dispatch Performance — part II: Typecases 104
A.3 Double Dispatch Performance — part III: Type Fields via Accessor 105
A.4 Double Dispatch Performance — part IV: Type Fields via getfield 105
A.5 Multi-Dispatch Performance 106

Chapter 1

Introduction

Object-oriented programming languages are the latest in a sequence of tools that expand the realm of practical and feasible applications. Stronger abstraction enables software designers to represent ever more complex problems. Encapsulation allows analysts to develop prototypes which concentrate on critical software requirements. Extensibility allows programmers to develop larger applications in less time by reusing existing components. Object-oriented programming languages provide greater expressiveness, and allow us to effectively solve a wider array of larger and more complex problems.

Object-oriented programming is built on the intuition of message-passing among collaborating entities. In this paradigm, one object synchronously requests a service from another object by sending it a message describing the desired service and containing additional information needed to complete the request. The receiving object responds by locating and executing a specific routine chosen to fulfill the service request. The object executing the service routine may send additional messages requesting further services from other objects in order to complete the original request.

Although the message may include arguments, the concept of a single, message-receiving object is implicit in this scheme. The naïve intuition underlying object-oriented programming is that *dispatch*, the process of selecting program code to service a message request, is based upon the message name and the type of the single receiver only. A more sophisticated dispatch process might consider the types of the other arguments included with the message when choosing the service code.

Indeed, it is all too common for the desired service to be dependent on the types of arguments as well as the single receiver. We see examples of this in graphical user interfaces, where program actions depend on the displayed object and the user action; in fundamental algorithms, where the cost to merge two lists depends on whether they are sorted or not; and in publish-subscribe applications, where the action depends on the subscribing client type and the kind of information published. The narrow focus of existing object-oriented languages onto a single receiver does not permit multiple objects to collectively and directly determine the dispatch decision.

In an attempt to escape the constraints of single-receiver dispatch, existing object-oriented languages permit the programmer to specify many service routines with the same name, each taking a different collection of parameters. This facility for *overloading* message names eases the single-receiver limitation, but does not eliminate it entirely. For example, print routines may be written for a variety of document types and hardware devices, but until the user selects a document to print and a destination device, the desired service routine is not known. A programmer writing the code to enable a print function can only dispatch on either the document object or the device object. Single-receiver dispatch cannot directly express this binary operation and the developer must work around this limitation.

Several work-arounds have been devised. Each of these solutions require the programmer to write code that sends a cascade of multiple single-receiver messages. In addition to the overhead of sending many messages, these alternative approaches incorporate type-specific information into customized program routines, resulting in increased programming time, reduced extensibility, and higher debugging costs.

A more general approach is to have the programming system recognize the types of each object collaborating in the message, and dispatch directly to a service routine appropriate to all of these types. The compile-time version of this, static multiple dispatch, operates with the declared types of the collaborating objects and forms the foundation for the ubiquitous method-overloading. The execution-time version, dynamic multiple dispatch, works with the actual object types present at the call site, and has been the subject of much recent research. A clear account of the interactions between type-checking and multiple dispatch provides a solid formal foundation for applying this technique. Efficient high-performance algorithms for dispatching multiple-receiver messages have been devised, and demonstrated in micro-benchmarks.

Multiple dispatch also eliminates the many ad-hoc dispatch work-arounds. The resulting programs are shorter, have better modularity, enhanced understandability, and greater robustness to change. In addition to removing the limitations of hand-coded routines, applying multiple dispatch decreases development time, reduces program errors, and improves extensibility. These benefits to the programmer have been demonstrated in research languages such as Cecil, CLOS, and Dylan.

However multiple dispatch is not part of mainstream object-oriented programming languages such as C++ and Java. Therefore, the performance differential between multiple dispatch and custom-written work-arounds has not been measured for commercial-grade applications and libraries. Also, the interaction between real-world language features — method visibility levels; class, instance, and interface methods; dynamic class-loading; reflection — and multiple dispatch has not been investigated. Finally, the potential incompatibilities between existing single-dispatch programs and extended multiple dispatch facilities have not been explored. Much is not yet understood about multiple dispatch in a production environment.

1.1 Goal

This dissertation describes the design, implementation, and evaluation of dynamic multiple dispatch in Java, a mainstream object-oriented programming language.

1.2 Terminology

Over the last two decades, a variety of terms have been coined to describe single and multiple dispatch in object-oriented programming languages. In particular, the term *multiple dispatch* does not effectively distinguish between making a single dispatch decision based on multiple arguments, and applying multiple single-receiver dispatches to achieve the same result. To avoid confusion, we will use the following three terms:

uni-dispatch denotes method selection based on a single, distinguished receiver;

double dispatch denotes a *sequence* of uni-dispatches used to locate and invoke a method based upon the types of multiple arguments; and

multi-dispatch denotes a *single* method selection decision which involves additional arguments as well as the single-receiver.

We will use the generic term **multiple dispatch** sparingly, only where we mean the final effect of either double dispatch or a multi-dispatch.

1.3 Organization

To achieve this goal, Chapter 2 begins by examining the characteristics of object-oriented programming languages, and recognizing that dispatch is the mechanism that gives these languages their expressive power. Next, we continue by identifying the two dimensions of dispatch: *time* and *arity*. We recognize that object-oriented programming languages already provide multi-dispatch in their method overloading algorithms, but this is a compile-time operation, hence involving static types. At execution-time most OO languages dispatch on the type of only a single argument. This differing dispatch arity (i.e. the number of arguments whose types are involved) is a key focus, and we explore uni-dispatch and its limitations. After demonstrating that dynamic multiple dispatch is required in common programming situations, we illustrate and critique the four double dispatch idioms used to circumvent uni-dispatch language restrictions. Next, we develop multi-dispatch as a natural extension to uni-dispatch and demonstrate the clarity, extensibility, and simplicity it brings to the problematic situations by eliminating programmer-written double dispatch.

Chapter 3 provides an introduction to the Java[1] programming language, and the Java Virtual Machine (JVM) that hosts it. We describe the binary format of compiled Java programs and the mechanism the JVM applies to effect dispatch. By considering the actions of the compiler and the virtual machine, we will develop crucial insights into how Java language features such as visibility and dynamic class loading interact with dispatch. Our review concludes with a description of the internal operation of the Research Virtual Machine, from Sun Microsystems, which we have extended to incorporate native multi-dispatch.

Our discussion proceeds in Chapter 4 to consider multi-dispatch for Java. We begin by re-examining Java dispatch constructs — method overloading and overriding — and we derive an analogous semantics for multi-dispatch that encompasses the range of Java language features, including type-safety, visibility, and reflection. In particular, we recognize a number of restrictions on the definition of multi-methods, and illuminate areas where uni-dispatch and multi-dispatch results can differ. Lastly, we identify a number of dispatch situations that must be accurately handled by a multi-dispatch Java.

Chapter 5 continues our exploration of multi-dispatch for Java by describing the details of our implementations. We begin by showing how to denote multi-dispatch in Java without changing language syntax. This notation also allows us to maintain existing uni-dispatch performance and binary compatibility with existing libraries and applications. Next, we illustrate how to extend the Research Virtual Machine to directly execute multi-dispatch. This is done in a modular fashion, isolating the multi-dispatch routine from the original virtual machine.

This modularity allows us to implement two different multi-dispatch algorithms. The first virtual machine acts as a reference platform, implementing a dynamic version of the static multi-dispatch algorithm corresponding to method overloading in Java. The other implementation is an extended custom Single Receiver Projections dispatcher that efficiently handles multi-dispatch with null arguments, lazy class-loading, and array types.

We continue our examination of multi-dispatch for Java in Chapter 6 with an evaluation of the compatibility, correctness, and performance of our multi-dispatch JVM. We show the compatibility and correctness of our implementations with existing uni-dispatch applications, and with semantics-checking multi-dispatch tests. We demonstrate performance aspects of our implementations using a number of micro-benchmarks — in particular, our tuned dispatcher is shown to be competitive with existing double dispatch in ordinary Java. Finally, we prove the viability of multi-dispatch Java by converting large, double-dispatch-intensive, application libraries, `Swing` and `AWT`, to use multi-dispatch. We quantify the software engineering benefits realized by this conversion, and demonstrate that multi-dispatch Java performs without application changes and without performance losses.

---

[1] Java is a registered trademark of Sun Microsystems Inc.

Chapter 7 closes the dissertation with a discussion of the potential for multi-dispatch in a production language. We begin by reviewing previous attempts, and show that most essentially automate an existing uni-dispatch work-around in semantically-simple situations. By scaling other techniques in the Dispatch Table Framework against our native Single Receiver Projections dispatcher, we conclude that a commercial multi-dispatch system could effectively replace double dispatch in standard Java. We also identify remaining open questions, and suggest some areas for performance improvement.

1.4 Contributions

The research contributions of this dissertation are:

1. The design and implementation of an extended Java Virtual Machine compiler that supports arbitrary-arity multi-dispatch with the properties:
    - (a) The Java syntax is not modified.
    - (b) The Java compiler is not modified.
    - (c) The programmer can select which classes and methods should use multi-dispatch.
    - (d) The performance and semantics of uni-dispatched methods are not affected.
    - (e) The existing class libraries are not affected.
2. A statement of the extended semantics of the Java Programming Language and Java Virtual Machine to accommodate multi-dispatch.
3. The introduction of a dynamic version of Java's static multi-dispatch algorithm.
4. The first head-to-head comparisons of table-based multi-dispatch techniques in a mainstream language.
5. The demonstration that multi-dispatch is competitive with more limited, verbose, and error-prone double-dispatch techniques.

Chapter 2

Object-Oriented Languages and Dispatch

Object-oriented (OO) languages provide powerful tools for expressing computations. One key abstraction is the concept of a *type hierarchy* which describes the relationships among types. Objects represent instances of these different types. Most existing object-oriented languages require each object variable to have a programmer-assigned *static type*. The compiler uses this information to recognize some coding errors. The *principle of substitutability* mandates that in any location where type `T` is expected, any sub-type of `T` is acceptable. Substitutability allows that object variable to have a different (but related) *dynamic type* at runtime.

Another key facility found in OO languages is method selection based upon the types of the arguments. This method selection process is known as *dispatch*. It can occur at compile-time, where only the static type information is available, and is known as *static dispatch* or method overloading. Dispatch can also occur at execution-time, where it is known as *dynamic dispatch* or method overriding. Dispatch, in both forms is leveraged by object-oriented languages to provide polymorphism — the execution of type-specific program code.

We can divide OO languages into two broad categories based upon how many arguments are considered during dispatch. *Uni-dispatch* languages select a method based upon the type of one distinguished argument; *multi-dispatch* languages consider more than one, and potentially all, of the arguments at dispatch time. For example, Smalltalk [21] is a uni-dispatch language. CLOS [40] and Cecil [9] are multi-dispatch languages.

2.1 Uni-Dispatch

C++ [16, 43, 44, 46, 45] and Java [22] are dynamic uni-dispatch languages. However for both languages, the compiler considers the static types of all arguments when compiling method invocations. Therefore, we can regard these languages as supporting static multi-dispatch.

Figure 2.1 depicts both dynamic uni-dispatch and static multi-dispatch in Java.

```
class Point {
 int x, y;
 void draw(Canvas c) { // Point-specific code }
 void translate(int t)           { x+=t; y+=t; }
 void translate(int tX, int tY) { x+=tX; y+=tY; }
}
class ColorPoint extends Point {
 Color c;
 void draw(Canvas C) { // ColorPoint code }
}
// same static type, different dynamic types
Point Pp = new Point();
Point Pc = new ColorPoint();
// static multi-dispatch
Pp.translate(5);   // one int version
Pp.translate(1,2); // two int version
// dynamic uni-dispatch
Pp.draw(aCanvas); // Point.draw(...)
Pc.draw(aCanvas); // ColorPoint.draw(...)
```

Figure 2.1: Dispatch Techniques in Standard Java

The Container Problem

Uni-dispatch limits the method selection process to consider the dynamic type of only a single argument, usually called the *receiver*. This is a substantial limitation: many methods accept additional arguments and the desired result depends not only on the receiver type, but on the other arguments as well.

Consider a situation where a collection of shapes (triangle, square) need to be sorted (see Figure 2.2). The desired result lists all triangles first, then all squares. However when a shape is placed into the collection and later accessed, its precise type (triangle or square) may be lost. Specifically if an object from the container is used as an argument in a message like `compareTo()`, its dynamic type is ignored. This loss of precise type information is known as the *container problem* [6].

If an object is used as the receiver of a method invocation, then it will regain its fully precise, dynamic type. So, the receiver of the `compareTo()` method, `shapes[i]`, will be correctly recognized as a `Triangle` or a `Square`. However the argument, `shapes[j]` will be dispatched as a `Shape` only. In other words, uni-dispatch does not automatically perform "reverse polymorphism" [6] on arguments.

As a result, the programmer must write special code to regenerate or verify this type information. That code, because it decides flow of control based on types, is a custom-written dispatcher. In the next section, we will review common programming idioms for writing these dispatchers.

```
abstract class Shape           { ... }
class Triangle extends Shape { ... }
class Square extends Shape   { ... }
class Main {
 static public void main(String args[]) {
  Shape shapes[] = { new Triangle(), new Circle(),
          new Circle(), new Square(), new Triangle() };
    // sort the vector of shape items
  for (int i=0; i<shapes.length; i++)
   for (int j=1; j<shapes.length; j++)
    if (GREATER == shapes[i].compareTo(shapes[j])
     swap(shapes[i], shapes[j]);
  printShapes(shapes);
 }
}
```

Figure 2.2: The Container Problem

2.2 Double Dispatch

Double dispatch occurs when a method explicitly checks an argument type and executes different code as a result of this check. Double dispatch is illustrated in Figure 2.3(a) (from Sun Microsystems `AWT` classes) where the `processEvent(AWTEvent)` method must process events in different ways, since event objects are instances of different classes. Since all of the events are placed in a queue whose static element type is `AWTEvent`, the compiler loses the more specific dynamic type information. When an element is removed from the queue for processing, its dynamic type must be explicitly checked to select the appropriate action. This is another example of the well-known container problem.

Double dispatch suffers from a number of disadvantages. First, double dispatch has the overhead of invoking a second method. In this example, the penalty is reduced because only one argument and no return values are involved. Second, the double-dispatch program is longer and more complex; this provides more opportunity for coding errors. Third, the double-dispatch program is more difficult to maintain since adding a new event type requires not only the code to handle the new event, but another cascaded `else if` statement.

The need for double dispatch develops naturally in several common situations. We will briefly examine four: binary operations, drag-and-drop, event-driven programming, and publish-subscribe. At the same time, we will recognize the different implementations of double-dispatch and review their deficiencies.

2.2.1 Binary Operations

The first example is binary operations [5], such as the `compareTo(Object)` method defined in interface `Comparable`. That method, when applied to a pair of objects, indicates that the first object is less-than (equal-to, greater-than) the second object by returning an integer less-than (equal-to, greater-than) zero. Any class implementing this interface gains a partial

```
package java.awt;
class Component {
  // double dispatch events to subComponent
  void processEvent(AWTEvent e) {
    if (e instanceof FocusEvent) {
      processFocusEvent((FocusEvent)e);
    } else if (e instanceof MouseEvent) {
      switch (e.getID()) {
      case MouseEvent.MOUSE_PRESSED:
        ...
      case MouseEvent.MOUSE_EXITED:
        processMouseEvent((MouseEvent)e);
        break;
      case MouseEvent.MOUSE_MOVED:
      case MouseEvent.MOUSE_DRAGGED:
        processMouseMotionEvent((MouseEvent)e);
        break;
      }
    } else if (e instanceof KeyEvent) {
      processKeyEvent((KeyEvent)e);
    } else if (e instanceof ComponentEvent) {
      processComponentEvent((ComponentEvent)e);
    } else if (e instanceof InputMethodEvent) {
      processInputMethodEvent((InputMethodEvent)e);
    }
    ...
  }
  void processFocusEvent(FocusEvent e)
    { ... }
  void processMouseEvent(MouseEvent e)
    { ... }
  void processMouseMotionEvent(MouseEvent e)
    { ... }
  void processKeyEvent(KeyEvent e)
    { ... }
  void processComponentEvent(ComponentEvent e)
    { ... }
  void processInputMethodEvent(InputMethodEvent e)
    { ... }
}
```

(a) Double Dispatch in Java

```
package java.awt;
class Component {

  void processEvent(AWTEvent e) { ... }


  void processEvent(MouseEvent e) {
    switch (e.getID()) {
    case MouseEvent.MOUSE_PRESSED:
      ...
    case MouseEvent.MOUSE_EXITED:
      processMouseEvent((MouseEvent)e);
      break;
    case MouseEvent.MOUSE_MOVED:
    case MouseEvent.MOUSE_DRAGGED:
      processMouseMotionEvent((MouseEvent)e);
      break;
    }
  }








  void processEvent(FocusEvent e)
    { ... }
  void processMouseEvent(MouseEvent e)
    { ... }
  void processMouseMotionEvent(MouseEvent e)
    { ... }
  void processEvent(KeyEvent e)
    { ... }
  void processEvent(ComponentEvent e)
    { ... }
  void processEvent(InputMethodEvent e)
    { ... }
}
```

(b) Equivalent Code in Multi-Dispatch Java

Figure 2.3: Double vs. Multi-Dispatch in Java

order by pairwise application of the interface method `compareTo(Object)`.

In Figure 2.4, we see an example of the `Comparable` interface implemented for the `Point` and `ColorPoint` classes introduced above. In this case, the programmer implemented double-dispatch using a *typecase* [1, 40] — a sequence of `instanceof` tests for type membership which select which type-specific code to execute.

Using typecases in Java reveals several defects which make code more error-prone and maintenance more expensive:

1. they increase code maintenance costs: as new types are added to the program, typecases must be located in the program, verified against the new types, and extended with additional type-specific clauses if necessary;
2. they introduce additional program code, which must then be tested and debugged;
3. they introduce additional dependencies on the type hierarchy of a program: most specific types must be tested first, most general types last.

```java
class Point implements Comparable {
 int x, y;
 int compareTo(Point p) {
  return ...
 }
 public int compareTo(Object o) {
  if (o instanceof Point) {
   return this.compareTo((Point) p);
  } else { // not a point
   return 0;
  }
 }
}
class ColorPoint extends Point {
 Color c;
 int compareTo(ColorPoint cp) {
  int r = super.compareTo(cp);
  if (0 == r) {
   return (this.c).compareTo(cp.c);
  } else {
   return r;
  }
 int compareTo(Object o) {
  if (o instanceof ColorPoint) { // first
   return this.compareTo((ColorPoint) o);
  } else if (o instanceof Point) { // second
   return this.compareTo((Point) o);
  } else {
   return super.compareTo(o);
  }
 }
}
```

(a) Typecases in Java

```java
class Point implements Comparable {
 int x, y;
 int compareTo(Point p) {
  return
 }
 public int compareTo(Object o) {
  return 0;
 }
}


class ColorPoint extends Point {
 Color c;
 int compareTo(ColorPoint cp) {
  int r = super.compareTo(cp);
  if (0 == r) {
   return (this.c).compareTo(cp.c);
  } else {
   return r;
  }
}
```

(b) Binary Operations in Multi-Dispatch Java

Figure 2.4: Typecases / Binary Operations

Typecases require some care in coding since all of the potential types must be identified. This is a common enough problem that at least one language, CLOS, includes a specific `etypecase` construct that exhaustively verifies that there is a case for each type. Equally important, the type tests must be performed with the most specific type first, and the most general last. For example, if `ColorPoint.compareTo(Object)` had tested for type `Point` first, then the `ColorPoint`-specific code would never be executed. Therefore, typecases can be a source of insidious bugs: no error is reported; the problem is only detected when an incorrect final result is observed.

From a software engineering standpoint, typecases introduce more places in the code that know about the relationship among program types. Clearly the programmer must indicate this in the declaration of the program classes — `ColorPoint` is a subclass of `Point`, but it is present in the ordering of typecases as well — `ColorPoint` must be tested before `Point`. Having this knowledge of the type-hierarchy in many places throughout the program increases the brittleness of the software.

Many other binary operations are immediately obvious: equality testing, arithmetic (especially with a variety of number representations - integer, floating-point, etc.), concatenation and merging (of lists, strings, sets, and so forth). One common implementation technique for these binary operations is typecases; virtually every OO language offers a type-testing construct. Indeed, some languages, e.g. Theta, UFO, and CLOS, include the entire typecase as a single construct.

2.2.2 Drag and Drop

Another common use for double dispatch is in drag-and-drop applications, where the result of a user action depends on both the data object dragged and on the target object. The `java.awt.dnd` package is dedicated to supporting the ability to drag data-source objects onto data-target objects, whereupon the target takes a source-type specific action. As a simple example, one might consider a user interface where documents and printers are represented. Documents are of various kinds: text, spreadsheet, presentation graphics. Printers also have differing capabilities: color versus monochrome for instance. When a user drags a given document onto the printer, the correct routines for rendering the given document type for the given printer need to be invoked.

A common implementation for this is call-backs through a delegate who provides a standard interface. For drag-and-drop printing, the printer would create a `Graphics` inner-class which implements the printer-specific operations in terms of a generic `Graphics` interface. The document object's `print` routine would be given this printer-specific delegate as the destination to render on. The print routine would dispatch back to the graphics object's generic routines: `drawLine()`, `drawText()`, etc.

This double dispatch solution is a standard example of the *Strategy* pattern [20]. It offers the advantage of interchangeable modules: new document types and printer types simply need to conform to the `Graphics` interface. In contrast to typecases, the print operation no longer contains knowledge of the source and target types. However there is a cost to this flexibility:

1. the code is longer and more complex — adding the subtlety of anonymous inner classes,

2. slower — involving extra dispatches, and

3. and less optimized — the generic interface may not apply printer-specific capabilities.

In some cases, the flexibility warrants the extra penalties, but in others, this is simply a work-around for the lack of multi-dispatch.

2.2.3 Event-Driven Programming

Our third example of common double-dispatch situations is event-driven programming. The prototypical example is graphical user interfaces.

As we saw in Figure 2.3, applications are written using base classes such as `Component` and `Event`, but we need to take action based upon the specific types of both `Component` and `Event`. Event-driving programming is the foundation of modern user interfaces. In this area, performance is paramount, and some type-hierarchies are well understood. In this case, a

```
interface Graphics {
 void drawText(String s);
 void drawPolyLine(Point p[]);
  ...
}
class MonoPrinter {
 void print(Document d) {
  d.printOn(new Graphics() {
   // Mono printer-specific
   void drawText(String s) { ... }
   void drawPolyLine(Point p[]) { ... }
   ...
  });
 }
}
class ColorPrinter {
 void print(Document d) {
  d.printOn(new Graphics() {
   // Color printer-specific
   void drawText(String s) { ... }
   void drawPolyLine(Point p[]) { ... }
   ...
  });
 }
}
interface Document {
 printOn(Graphics g);
}
class Spreadsheet implements Document {
 void printOn(Graphics g) {
  g.drawPolyLine(...);
  g.drawText(...);
   ...
 }
}
class TextDoc implements Document {
 void printOn(Graphics g) {
  g.drawPolyLine(...);
  g.drawText(...);
   ...
 }
}
```

(a) Strategy Pattern in Java

```
class MonoPrinter {
  // optimised routines
 void print(SpreadSheet s) { ... }
 void print(TextDoc t) { ... }
  ...
}

class ColorPrinter {
  // optimised routines
 void print(SpreadSheet s) { ... }
 void print(TextDoc t) { ... }
  ...
}

class Spreadsheet { ... }

class TextDoc { ... }
```

(b) Drag and Drop in Multi-Dispatch Java

Figure 2.5: Strategy Pattern / Drag and Drop

common technique is to use a single class, with program-specific *type fields* to represent variants.[1] We see this in the previous example for `MouseEvents`, where manifest constants such as `MouseEvent.MOUSE_PRESSED` and `MouseEvent.MOUSE_EXITED` are used to distinguish the kind of mouse event that occurred.

An equivalent program, partially using multi-dispatch, would resemble Figure 2.3(b). For clarity, we did not completely convert the code to use multi-dispatch; we maintained the numeric case statement and double dispatch to select among `MouseEvent` categories. The more complete factoring in Figure 2.6 of `MouseEvent` into `MouseButtonEvent` and `MouseMotionEvent` would eliminate the remaining double dispatch, resulting in a *Fully Multi-Dispatch* version of the code. The dynamic multi-dispatcher will select the correct method at runtime based upon the *dispatchable arguments* in addition to the *receiver argument* (the instance of `Component`). Individual component types can still override the methods that accept specific event types (e.g. `KeyEvent`, `FocusEvent`) and will do so without invoking the double-dispatch

---

[1] The term type field was coined by Stroustrup [44], Section 6.2.4.

```
// introduce new subtypes rather than use numbers
class MousePressedEvent extends MouseEvent { ... }
class MouseExitedEvent extends MouseEvent { ... }
class MouseMovedEvent extends MouseEvent { ... }
class MouseDraggedEvent extends MouseEvent { ... }

// rename methods
class Component {
  void processEvent(MouseEvent e) { } // entpy
  void processEvent(MousePressedEvent e) {
    ...// used to be processMousePressedEvent()
  }
  void processEvent(MouseExitedEvent e) {
    ...// used to be processMouseExitedEvent()
  }
  void processEvent(MouseMovedEvent e) {
    ...// used to be processMousePressedEvent()
  }
  void processEvent(MouseDraggedEvent e) {
    ...// used to be processMousePressedEvent()
  }
}
```

Figure 2.6: Multi-Dispatch Event-Driven Programming without Type Fields

code.

Type fields operate in the same way as the typecases we saw previously, except that they can use a much faster numeric switch statement. What they gain in performance is traded off against flexibility and maintainability. Stroustrup [44], notes that type fields are "an error-prone technique that leads to maintenance problems ... the use of type fields is a violation of modularity and data hiding." The type fields must be consistent in all uses, implying a single centrally-maintained list.

This is clearly illustrated by the following incompatibility note accompanying the just-released Java 1.4 platform [47]:

> The value of static final field `MOUSE_LAST` in class `java.awt.event.MouseEvent` has changed to 507 beginning in J2SE 1.4.0. In previous versions of the Java 2 Platform, the value of `MOUSE_LAST` was 506.
>
> Because compilers hard-code static final values at compile-time, code that refers to `MOUSE_LAST` and that was compiled against a pre-1.4.0 version of `java.awt.event.MouseEvent` will retain the old value. Such code should be re-compiled with the version 1.4.0 compiler in order to work with J2SE 1.4.0.

. Of course, any such recompilation makes the code incompatible with Java 1.3 or below. Even worse, if this value is used inside of a commercial library to which you do not have source, then you cannot recompile. Last, once the original program is extended, each code path through the switch must be tested to ensure no programming errors have crept in.

2.2.4 Publish-Subscribe

A fourth example where double-dispatch occurs is *Publish-Subscribe* designs, where publisher classes maintain a list of subscribers, and push items to their subscribers. Again, the recurring difficulty is that different kinds of publishers and subscribers exist and the correct handling of items depends on both types.

```
abstract class Publisher {
 Subscriber subscribers[];
 abstract method send(Subscriber s);
 method publish(Item item) {
  for (int i=0; i<subscribers.length; i++)
   this.send(subscribers[i], item);
 }
}
// these routines inform compiler
// of specific Publisher type
class Pub1 extends Publisher {
 void send(Subscriber s, Item item) {
  // re-dispatches to Subscriber.recv(Pub1)
  s.recv(this, item);
 }
}

class Pub2 extends Publisher {
 void send(Subscriber s, Item item) {
  // re-dispatches to Subscriber.recv(Pub2)
  s.recv(this, item);
 }
}
abstract class Subscriber {
 abstract void recv(Pub1 p, item);
 abstract void recv(Pub2 p, item);
}
class Sub1 extends Subscriber {
 void recv(Pub1 p, Item item) { /* S1xP1 */ }
 void recv(Pub2 p, Item item) { /* S1xP2 */ }
}
class Sub2 extends Subscriber {
 void recv(Pub1 p, Item item) { /* S2xP1 */ }
 void recv(Pub2 p, Item item) { /* S2xP2 */ }
}
```

(a) Visitor Pattern in Java

```
abstract class Publisher {
 Subscriber subscribers[];
 abstract void send(Subscriber s);
 method publish(Item item) {
  for (int i=0; i<subscribers.length; i++)
   this.send(subscribers[i], item);
 }
}


class Pub1 extends Publisher {
 void send(Sub1 s, Item item) { /* S1xP1 */ }
 void send(Sub2 s, Item item) { /* S2xP1 */ }
}



class Pub2 extends Publisher {
 void send(Sub1 s, Item item) { /* S1xP2 */ }
 void send(Sub2 s, Item item) { /* S2xP2 */ }
}


abstract class Subscriber { ... }



class Sub1 extends Subscriber {
 ...
}

class Sub2 extends Subscriber {
 ...
}
```

(b) Publish Subscribe in Multi-Dispatch Java

Figure 2.7: Visitor Pattern / Publish Subscribe

In this case, we show Publish-Subscribe using the *Visitor* pattern [20]. The Visitor pattern is not novel; it was first published in 1986 [27], but had been applied extensively before that. In this pattern, the publisher uni-dispatches to a publisher-specific send routine, at which point, the publisher type is known. This publisher-specific method then encodes the publisher type into the method signature of a second uni-dispatch to a subscriber-specific receive routine. At this second dispatch, both the publisher-type and subscriber-type are known and the correct operation executed.

It is important to note that Java does not require the publisher-type to be encoded into the method name, as other languages such as Smalltalk do. The compiler knows the specific type of `this` for each `PubN.send()` method, and encodes it into the method signature. However the visitor pattern still suffers from drawbacks:

1. additional program code must be written — more code to test and debug;

2. there is the overhead of a second dispatch — increasing program execution time
3. replication of type structure — every publisher type must be encoded into the abstract `Subscriber` class as well as in each subscriber.

2.3 Multi-Dispatch

In each of the previous situations, we supplied a multi-dispatch version of the code. This multi-dispatch version overcomes many of the deficiencies identified.

1. In each case, the multi-dispatch version is shorter. There are fewer lines of code, reducing coding time, debugging effort, and maintenance costs.
2. In each case, the multi-dispatch version is clearer. Type specific routines are gathered into fewer places, and laid-out as individual routines.
3. Knowledge of the types involved is localized only to the places that actually depend on it: the class definitions and method definitions. In particular, `instanceof` tests and "magic" constants are eliminated along with their potential for introducing and masking errors.

However the multi-dispatch versions require specialized support for dispatching on the additional arguments. Before we examine how to extend Java to support the multi-dispatch versions, we will first examine how uni-dispatch works in standard Java.

Chapter 3

Dispatch in Java

The Java Programming Language [22] is a static multi-dispatch, dynamic uni-dispatch, dynamic loading, object-oriented language. Java programs are compiled by `Javac` (or other compiler) into sequences of bytecodes — primitive operations of a simple stack-based computer. These bytecodes are interpreted by a JVM written for each hardware platform. Our work concentrates on the *classic* VM (now known as the *Research Virtual Machine*[1]) written in C and distributed by Sun Microsystems, Inc. Other JVM implementations exist and many include *Just-In-Time* (JIT) compiler technology to enhance the interpretation speed at runtime by replacing the bytecodes with equivalent native machine instructions.

Before we look at how to implement multi-dispatch in the virtual machine, we first need to understand the binary representation that the virtual machine executes, how method invocations are translated into the virtual machine code, and how the JVM actually dispatches the call-sites.

3.1 Java Classfile format

The JVM reads the bytecodes, along with some necessary symbolic information from a binary representation, known as a `.class` file. Each `.class` file contains a symbol table for one class, a description of its superclasses, and a series of method descriptions containing the actual bytecodes to interpret. This symbolic information provides the names necessary to bind separately compiled classes together at run-time. We leverage this symbolic information, called the *constant pool*, to implement multi-dispatch.

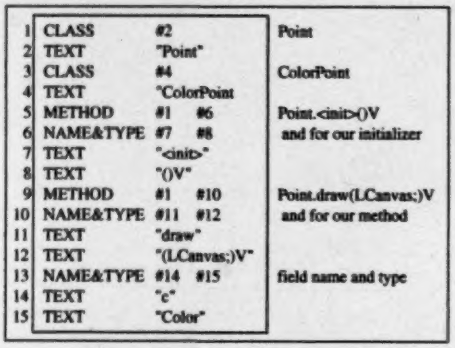
Figure 3.1 shows the layout of the constant pool for the `ColorPoint` class shown in Figure 2.1.

Conceptually, the constant pool consists of an array containing text strings and tagged references to text strings. In Figure 3.1, class `Point` is represented by a tag entry at location

---

[1] The Research Virtual machine was initially released as the *classic* reference VM. Sun Microsystems later renamed it the *Exact* VM. With the advent of the *HotSpot* VM, the classic VM was renamed again, becoming the *Research* VM.

1 that indicates that it is a CLASS tag and that we should look at constant pool location 2 for the name text. Then, the constant pool contains the text string "`Point`" at location 2. Therefore, a class symbol requires two constant pool entries. Method references are similar, except they require five constant pool entries.



1	CLASS	#2		Point
2	TEXT	"Point"		
3	CLASS	#4		ColorPoint
4	TEXT	"ColorPoint		
5	METHOD	#1	#6	Point.<init>()V
6	NAME&TYPE	#7	#8	and for our initializer
7	TEXT	"<init>"		
8	TEXT	"()V"		
9	METHOD	#1	#10	Point.draw(LCanvas;)V
10	NAME&TYPE	#11	#12	and for our method
11	TEXT	"draw"		
12	TEXT	"(LCanvas;)V"		
13	NAME&TYPE	#14	#15	field name and type
14	TEXT	"c"		
15	TEXT	"Color"		

Figure 3.1: A Simple Constant Pool for `ColorPoint` (Figure 2.1)

In our example, constant pool location 9 contains the tag declaring that it contains a METHOD. It references the CLASS tag at location 1, to define the static type of the class containing the method to be invoked. In this case, the class happens to be `Point` itself, but, more often, this is not the case. The METHOD entry also references the NAME-&-TYPE entry at location 10. This NAME-&-TYPE entry contains pointers to text entries at locations 11 and 12. The first location, 11, contains the method name, "`draw`". The second location, 12, contains an encoded signature "`(LCanvas;)V`" describing the number of arguments to the method, their types, and the return type from the method. In our example, we see one argument listed between the parenthesis, with classname "`Canvas`" (demarked by the characters `L` and `;`) and that the return type is `void` (denoted by the `V` after the parentheses).

3.2 Static Multi-Dispatch in `Javac`

The Java compiler converts source code into a binary representation. When it encounters a method invocation, `Javac` must emit a constant pool entry that describes the method to be invoked. It must provide an exact description, so that, for instance, the two `translate(...)` methods in `Point` can be distinguished at runtime. Therefore, it must examine the types of the arguments at a call-site and select between them. This selection process, which considers the static types of all arguments, can be viewed as a static multi-dispatch.

The Java Language Specification, 2nd Edition (JLS) [22] provides an explicit algorithm for static multi-dispatch called *Most Specific Applicable* (MSA). At a call-site, the compiler begins with a list of all methods implemented and inherited by the (static) receiver type. Through a series of culling operations, the compiler reduces the set of methods down to a single most specific method. The first operation removes methods with the wrong name, methods that accept an incorrect number of arguments, and methods that are not accessible from the call-site. This latter group includes private methods called from another class and protected methods called from outside of the package.

Next, any methods which are not compatible with the static type of the arguments are also removed. This test relies upon testing *widening conversions*, where one type T_{sub} can be widened to another T_{super} if and only if T_{sub} is the same type as T_{super} or a subtype of T_{super}. For example, a `FocusEvent` can be widened to an `AWTEvent` because the latter is a super-type of the former.[2] The opposite is not valid: an `AWTEvent` cannot be widened to a `FocusEvent`; indeed a type-cast from `AWTEvent` to `FocusEvent` would need to be a type-checked *narrowing* conversion.

Finally, `Javac` attempts to locate the single *most specific* method among the remaining subset of *statically applicable* methods. One method $\mathtt{M}(T_{1,1},\ldots,T_{1,n})$ is considered more specific than $\mathtt{M}(T_{2,1},\ldots,T_{2,n})$ if and only if each argument type $T_{1,i}$ can be widened to $T_{2,i}$ for each $(i = 1,\ldots,n)$, and for some j, $T_{2,j}$ cannot be widened to $T_{1,j}$. In effect, this means that any set of arguments acceptable to $\mathtt{M}(T_{2,1},\ldots,T_{2,n})$ is also acceptable to $\mathtt{M}(T_{1,1},\ldots,T_{1,n})$, but not vice versa.

Given the subset of applicable methods, `Javac` selects one M_t as its tentatively most specific. It then checks each other candidate method M_c by testing whether its arguments can be widened to the corresponding argument in M_t. If this is successful, then M_c is at least as specific as M_t; the compiler adopts M_c as the new tentatively most specific method — the method M_t is culled from the candidate list. If the first test, whether M_c be widened to M_t, is unsuccessful, then the compiler checks the other direction: can M_t be widened to M_c. If so, then the compiler drops M_c from the candidate list.

Unfortunately, both tests can fail. To illustrate this, consider the first two methods in Figure 3.2. The first argument of the first method (`ColorPoint`) can be widened to the type of the first argument of the second method (`Point`). However the opposite is true for the second argument of each method. If we invoke `colorBox` with two `ColorPoint` arguments, both methods apply. If the third method was not present, we would have an *ambiguous method* error from the compiler. The third method, taking two `ColorPoints`, removes the ambiguity because it is more specific than both of the other methods. It allows both of the others to be culled, giving a single most specific method.

Primitive types[3], when used as arguments, are tested at compilation time in the same way as other types. Primitive widening conversions in the Java Language are defined which effectively impose a standard type hierarchy on the primitive types. This type hierarchy for

---

[2] The JLS separately recognizes identity conversions (a `FocusEvent` can be converted into a `FocusEvent`). `Javac` does not distinguish them, so we do the same for our exposition.

[3] Java provides non-object types `byte(B)`, `char(C)`, `short(S)`, `int(I)`, `long(L)`, `float(F)`, `double(D)`, and `boolean(Z)`. These are called primitive types.

```
colorBox(ColorPoint p1, Point p2) { ... }
colorBox(Point p1, ColorPoint p2) { ... }
              // conflict method removes ambiguity
colorBox(ColorPoint p1, ColorPoint p2) { ... }
```

Figure 3.2: Ambiguous and Conflict Methods

primitives in the Java Language is shown in Figure 3.3.

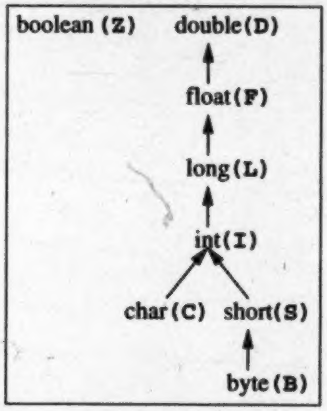


Figure 3.3: Hierarchy of Primitive Types in the Java Language

However it is important to recognize that these widening conversions provide only an "ad-hoc" polymorphism [7, 41]. The compiler emits the coercions, and from the JVM perspective, the original type is no longer visible. The JVM sees each type as independent, with no sub-typing relationships at all. Although the runtime implements the representation changes as bytecodes (`i2l`, `i2f`, etc.), it offers no equivalent automatic conversions. Once the compiler inserts widening casts as needed, the JVM no longer has access to the original primitive type as seen in Figure 3.4. The bytecode emitted for the two method invocations at the end of Figure 3.4 are identical.

This results from the compiler examining the methods for a receiver of static type `Super`, discovering no method accepting a `byte`, and automatically promoting the argument to an `int`. The NAME-&-TYPE becomes "`method`" and "`(I)V`" (one argument of `int` type, returning `void`. When uni-dispatch occurs, the method accepting an `int` is executed, even if the dynamic receiver has an implementation accepting a `byte` argument.

Widening casts of object types also lose precision as seen in Figure 3.5. However the actual object available at dispatch is self describing, it knows it is a `String`, so the JVM has the potential to regain the precise type — this is the essence of our Multi-Dispatch technique. It is important to note that the loss of precision with primitive types is irreversible.

Java supports arrays as a parameterized type.[4] Consider the declarations in Figure 3.6

---

[4] Sun Microsystems has endorsed a more general parametric type system for Java, described in JSR14 [48].

```
class Super {
  void method(int i) {
    System.out.println("Super.method(I)");
  }
}
class Sub extends Super {
  void method(byte b) {
    System.out.println("Sub.method(B)");
  }
  void method(int i) {
    System.out.println("Sub.method(I)");
  }
}
Super s1 = new Sub();
byte b = 0;
s1.method(b); // automatic promotion
                --> Sub.method(I)
Sub s2 = new Sub();
byte b = 0;
s2.method((int)b);
                --> Sub.method(I)
```

Figure 3.4: Automatic Primitive Promotions

which define a class, `Super`, and a subclass, `Sub`, and methods that accept as arguments: `Super`, `Sub`, `Super[]` (an array of `Super`) and `Sub[]` (an array of `Sub`). `Javac` provides static multi-dispatch of the array types that parallels the dispatch for the plain types. In particular, `Sub[]` is a treated as a subtype of `Super[]` exactly as `Sub` is a subtype of `Super`. In this way, Java arrays act in the same way as parameterized types such as templates do in C++. Dynamic multi-dispatch must provide the same semantics for arrays; hence dynamic multi-dispatch must implement a simple version of parametric polymorphism.

The Java Language specification [22] provides the type hierarchy illustrated in Figure 3.7. Essentially, for each array dimension, the type hierarchy parallels the "0-dimension" hierarchy for the bare types. These array hierarchies connect back to the next dimension

```
class Super {
  void method(Object o) {
    System.out.println("Super.method(Object)");
  }
}
class Sub extends Super {
  void method(String s) {
    System.out.println("Sub.method(String)");
  }
  void method(Object o) {
    System.out.println("Sub.method(Object)");
  }
}
Super s = new Sub();
String o = "string";
s.method(o); // automatic widening
                 --> Sub.method(Object)
```

Figure 3.5: Loss of Precision

down, through the recognition that each *slice* (the highest subscript) corresponds to a single object of types `java.io.Serializable` and `java.lang.Cloneable` with one-lower dimension. These two interfaces further connect to the `java.lang.Object` class at that one-lower dimension.[5] For example, using the previous class definitions, `Sub[][]` is a subtype of `Super[][]`, which is a subtype of `java.io.Serializable[]` and `java.lang.Cloneable`, which are both subtypes of `java.lang.Object[]`, which is a subtype of both `java.io.Serializable` and `java.lang.Cloneable`, which finally, are `java.lang.Objects`[6].

One important note is that the latent polymorphism present for primitive types does not carry across the array hierarchies. Bare primitive types have their representation widened (coerced), but arrays parameterized on those primitive types do not maintain the coercion-based subtype relationship.

In contrast, object types do retain their subtype relationship across parameterization to arrays. The JVM maintains a more consistent view of types; note that the independence of primitive types is mirrored in the type sub-hierarchies for each higher array dimension (see Figure 3.8). However the compiler is responsible for inserting an appropriate coercion operation, therefore the JVM does not need to concern itself with this latent polymorphism.[7]

---

[5] Anomalously, every interface considers `java.lang.Object` to be its direct superclass, even if it can trace a path through a superinterface to `java.lang.Object`.

[6] Henceforth, we will dispense with the package names, `java.lang`, `java.io`, etc., for system classes.

[7] By definition, latent polymorphism disappears before runtime [7], so it cannot affect dynamic dispatch.

```
class Super { ... }
class Sub extends Super { ... }
class Main {
  String method(Super s) { return "method(super)"; }
  String method(Sub s) { return "method(sub)"; }
  String method(Super s[]) { return "method(super[])"; }
  String method(Sub s[]) { return "method(sub[])"; }
  public static void main(String args[]) {
    Super super = new Super();
    Super sub1 = new Sub();
    Sub sub2 = new Sub();
    System.out.println(method(super));
    System.out.println(method(sub1));
    System.out.println(method(sub2));
    Super superA[] = new Super[1];
    Super subA1[] = new Sub[1];
    Sub subA2[] = new Sub[1]
    System.out.println(method(superA));
    System.out.println(method(sub1A));
    System.out.println(method(sub2A));
  }
}
                    --> method(super)
                    --> method(super)
                    --> method(sub)
                    --> method(super[])
                    --> method(super[])
                    --> method(sub[])
```

Figure 3.6: Arrays as Parameterized Types

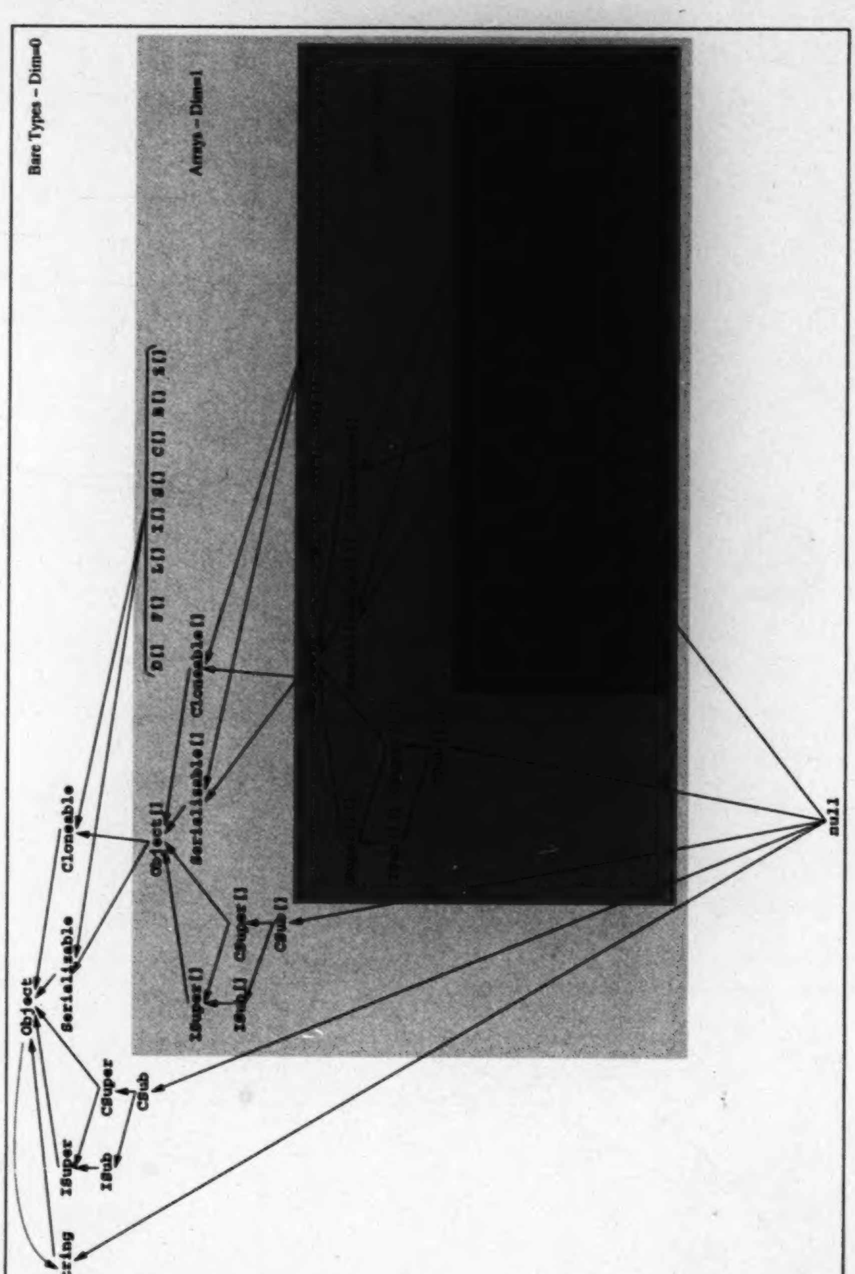
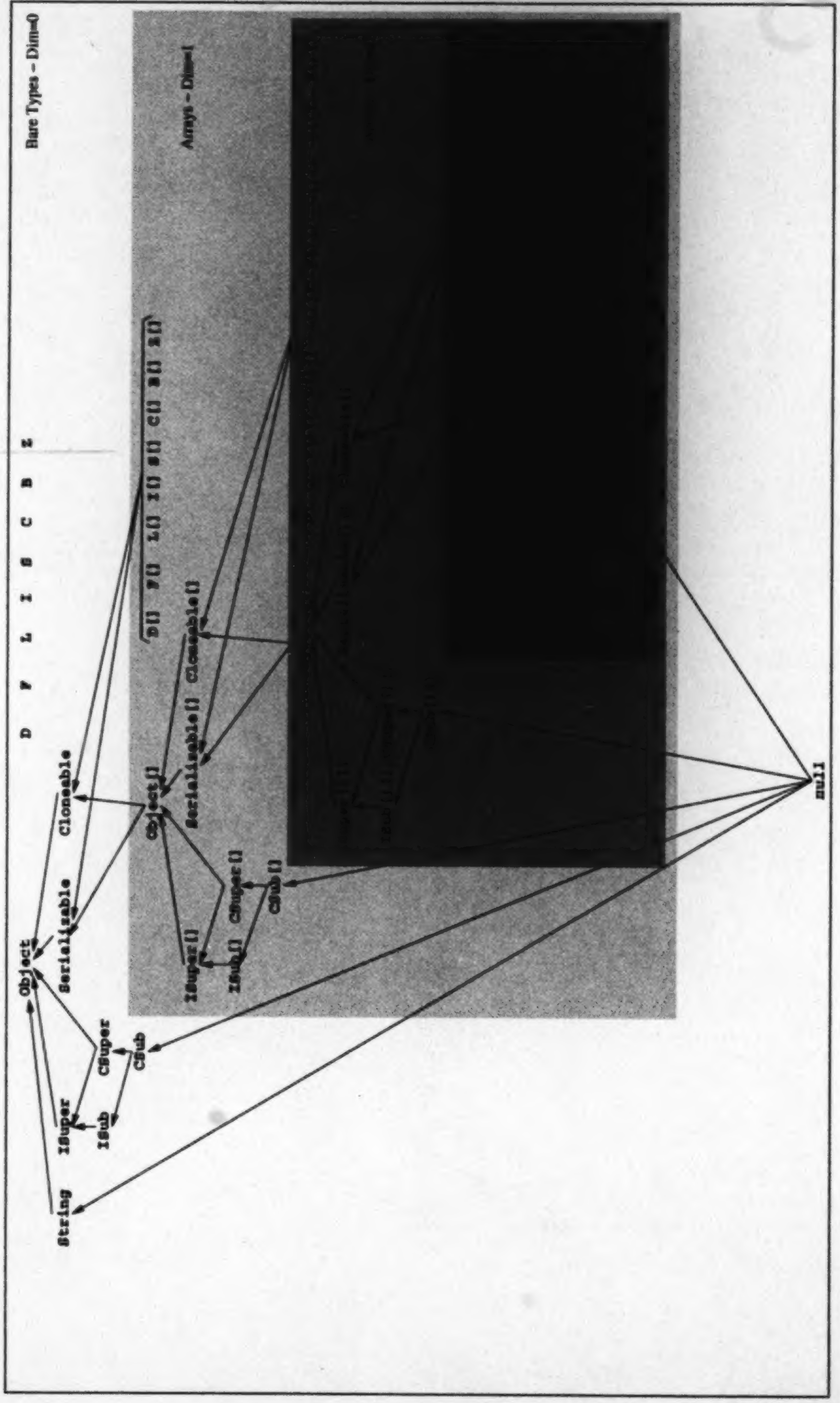

Bare Types – Dim=0
Arrays – Dim=1
D
F
L
I
C
S
B
String
ISuper
ISub
CSuper
CSub
Object
Serializable
Cloneable
Object[]
Serializable[]
Cloneable[]
D[] F[] L[] I[] S[] C[] B[] Z[]
ISuper[]
ISub[]
CSuper[]
CSub[]
null

Figure 3.7: Java Language Type Hierarchy



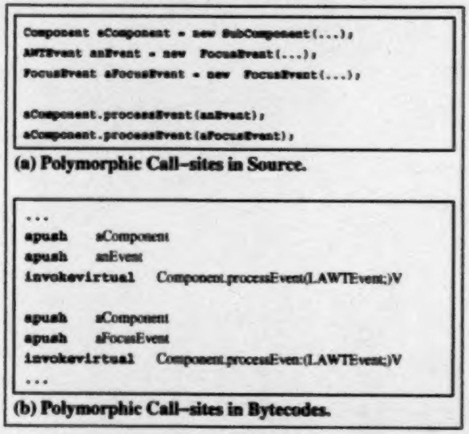
Bare Types - Dim=0
D F L I S C B Z
Object
Serializable
Cloneable
String
ISuper
ISub
CSuper
CSub
Arrays - Dim=1
D[] F[] L[] I[] S[] C[] B[] Z[]
Object[]
Serializable[]
Cloneable[]
ISuper[]
ISub[]
CSuper[]
CSub[]
null

Figure 3.8: Java Virtual Machine Type Hierarchy

These figures also illustrate one other subtlety in the Java Language type system (versus the JVM type system) involving the `String class`. Every reference type is convertible into a `String`, because of the `Object.toString()` method. The `Javac` compiler automatically inserts coercions (as calls to the `toString()` method) as needed. There is no automatic coercion of an array of `Objects` into an array of `Strings`. With regard to implementing Multi-Dispatch Java, the automatic coercion does not exist: the JVM sees a compiler-inserted method invocation.

3.3 Dynamic Uni-Dispatch in the JVM

Methods are stored in the `.class` file as sequences of virtual machine instructions. Within a stream of bytecodes, method invocations are represented by `invoke` bytecodes that occupy three bytes[8]. The first byte contains the opcode (e.g. `0xb6` for `invokevirtual`). The remaining two bytes form an index into the constant pool. The constant pool must contain a METHOD entry at the given index. This entry contains the static type of the receiver argument (as the CLASS linked entry), and the method name and signature (through the NAME&TYPE entry). Figure 3.9 shows the pseudo-bytecode[9] for invoking the method `Component.processEvent(AWTEvent)` twice. Surprisingly, the second call executes the same method, because `Component` does not define `processEvent(FocusEvent)`; static multi-dispatch locates `Component.processEvent(AWTEvent)` as the most specific applicable method.



```
Component aComponent = new SubComponent(...);
AWTEvent anEvent = new  FocusEvent(...);
FocusEvent aFocusEvent = new  FocusEvent(...);

aComponent.processEvent(anEvent);
aComponent.processEvent(aFocusEvent);
```

(a) Polymorphic Call–sites in Source.

```
...
apush         aComponent
apush         anEvent
invokevirtual   Component.processEvent(LAWTEvent;)V

apush         aComponent
apush         aFocusEvent
invokevirtual   Component.processEven:(LAWTEvent;)V
...
```

(b) Polymorphic Call–sites in Bytecodes.

Figure 3.9: Polymorphic Call-sites — two views

From the opcode, `invokevirtual(IV)`, the JVM knows that the next two bytes contain the constant pool index of a METHOD descriptor. From that descriptor, the JVM can locate the method name and signature. The JVM parses the signature to discover that the method to be invoked requires a receiver argument and one other argument. Therefore, the JVM peeks into the operand stack and locates the receiver argument. At this point, the JVM has the information it needs to begin searching for the method to invoke. The JVM has the name, the signature, and the receiver-type of the message.

---

[8] The `invokeinterface` bytecodes occupy 5 bytes.

[9] Rather than show constant pool indexes, we show their values directly.

For an `invokestatic` (IS) bytecode, the JVM does not have a receiver argument, but the METHOD descriptor gives a classname that should[10] contain the desired method. So, again the JVM has the same three pieces of information: the method name, the signature, and a "pseudo-receiver"-type. For an `invokespecial` (INV) bytecode, the NAME-&-TYPE references the receiver type in the CLASS part.

The invoke bytecode has been de-constructed and three crucial pieces of information are now available:

1. the class containing the method (the "receiver" type),
2. the name of the method, and
3. the signature of the method (encoding the number and types of the arguments, and the return type).

Now uni-dispatch proceeds in three steps: *resolution*, *lookup*, and *invocation*.

3.3.1 Resolution

As we saw in Section 3.1, method references are stored symbolically in the constant pool for a class. Before a method can be invoked, the symbolic reference must be "linked" to an actual method. In many respects, this process is identical to that of any other symbolic linker that binds multiple object files together. The JVM treats each class as its own dynamic link library, and demand loads it (and any other super-classes it depends upon). Symbolic linker technology to support this has been available for several years.

The JVM Specification (Section 5.4.3.3) [32] provides a recursive algorithm for *resolving* a method reference and locating the correct method: Beginning with the methods defined for the precise receiver argument type, scan for an exact match for the name and signature. If one is not found, search the superclass[11] of the receiver argument, continuing up the superclass chain until `Object`, the root of the type hierarchy, is searched. If an exact match is not found, throw an `AbstractMethodError`. This look-up process applies to each of the `invoke` bytecodes.

This lookup process implies that resolved methods may be inherited from super-classes; exactly what OO technology expects. However it also applies to static methods as well. Consider the code in Figure 3.10. Resolution in class `Main` of the method reference for

---

[10] Resolution, described below, can permit super-classes to contain the desired method as well.

[11] Java provides only single inheritance of program code.

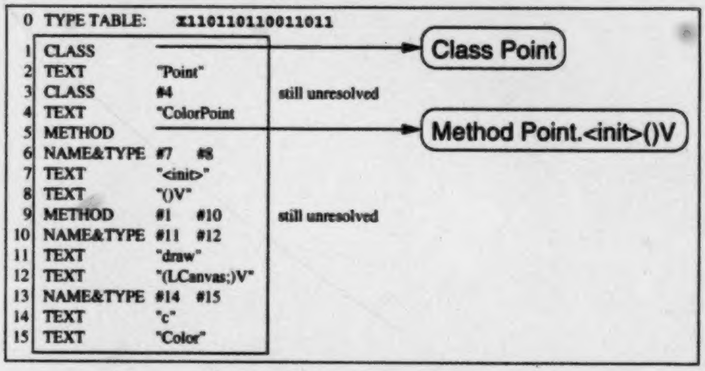
`Sub.method()` locates the implementation in class `Sub`. However if we remove that method from `Sub.Java` and recompile only that changed file, the JVM does not report a missing method. If we had recompiled `Main.Java`, the constant pool reference would have been changed by `Javac` to have NAME-&-TYPE as `Super.method()`. However without recompiling, the JVM resolves up through the superclass chain, locating `Super.method()` as having the correct name and type.

```
// -------- file Super.java -------
class Super {
  static void method() {
    System.out.println("Super.method()");
  }
}
// -------- file Sub.java -------
class Sub extends Super {
  static void method() {
    System.out.println("Sub.method()");
  }
}
// -------- file Main.java -------
class Main {
  static public void main(String args[]) {
    Sub.method();
  }
}
        --> Sub.method()

// remove Sub.method() and recompile Sub.java only
        --> Super.method()
```

Figure 3.10: Static Inheritance Through Resolution

This resolution process is a time-intensive operation. To reduce this latency, the resolved method is cached in the constant pool in place of the original method reference. This fact is encoded in the *type table* located at constant pool index 0. Figure 3.11 illustrates this.



0	TYPE TABLE:	X110110110011011	
1	CLASS		
2	TEXT	"Point"	
3	CLASS	#4	still unresolved
4	TEXT	"ColorPoint	
5	METHOD		
6	NAME&TYPE	#7 #8	
7	TEXT	"<init>"	
8	TEXT	"()V"	
9	METHOD	#1 #10	still unresolved
10	NAME&TYPE	#11 #12	
11	TEXT	"draw"	
12	TEXT	"(LCanvas;)V"	
13	NAME&TYPE	#14 #15	
14	TEXT	"c"	
15	TEXT	"Color"	

Figure 3.11: A Constant Pool After Method Resolution

The next time this method reference is applied by another `invoke` bytecode, the type table is checked, and the cached method is used directly. Sun Microsystems eliminates the need to examine the type table for each `invoke` bytecode by an extension known as *quick opcodes* described in Chapter 9 of the Java Virtual Machine Specification [31]. The process is quite straightforward: provide replacement `invoke` bytecodes that inform the JVM to avoid the resolution process.

Methods are only resolved when an `invoke` bytecode is being interpreted. Once the method is resolved, replace the bytecode instruction at that location with the replacement listed below:

- `invoke-virtual`(`IV`) is replaced by `invoke-virtual-quick-wide`(`IVQW`),
- `invoke-static`(`IS`) is replaced by `invoke-static-quick`(`ISQ`),
- `invoke-special`(`INV`) is replaced by an `invoke-nonvirtual-quick`(`INVQ`),[12] and
- `invoke-interface`(`II`) is replaced by `invoke-interface-quick`(`IIQ`).

There are more quick bytecodes that can be inserted, but they can only be inserted after lookup, which we examine next.

3.3.2 Lookup

Once a method is resolved, the next step for the JVM is *lookup*. This is the step which actually performs the dynamic dispatch; that is, where the JVM selects type-specific code to execute. After examining how the JVM stores methods, we will consider the lookup process for each `invoke` bytecode separately.

Every class in the Research JVM is represented by a `ClassClass` structure (a *classblock*) containing two lists of methods. Each list is represented by a pointer and an integer length.

For a given classblock,`cb`, the first array is located by the pointer `cbMethodTable(cb)` and contains `cbMethodTableSize(cb)` instances of `struct methodblock` (referred to as *methodblocks*). This *method table* contains all of the methods implemented by this class — every method that appears in the body of the `.class` source file.

The second list, located by `cbMethods(cb)` and of length `cbMethodsCount`, contains pointers to all of the methods inherited by and inheritable from this class. This list excludes private methods, static methods, and constructors, since none of these are inherited. Each of the positions in this array will be referred to as a *slot*, and given a index beginning with one.[13]

Also, this second method array consists of a list of methodblock pointers constructed by merging methods from the superclass and the current class in a "stable" way. Each pointer in this second array points to a methodblock in the current class or a superclass. A virtual method in the current class that has the same NAME-&-TYPE as an inherited method overwrites the inherited method's slot in the current class' `cbMethods` list. In this way, the slot number for any method (and its overriding implementations) is the same in all subclasses. In C++ parlance, this second method list is a ***virtual function table***.

---

[12] As we will see in Section 3.3.2, this bytecode invokes methods which are *non-virtual* hence the acronyms `INV` and `INVQ`.

[13] The JVM uses `(offset == 0)` as a special marker, so slot 0 is left unused.

`Invoke-static` and `Invoke-static-quick` Bytecodes

The `invoke-static` bytecode performs no lookup; its constant-pool reference is the desired method. This matches the semantics that static methods are neither inherited nor overridden — they are ***statically uni-dispatched***.

`Invoke-virtual` et al. Bytecodes

Methods that are not static, not private, and not constructors can be overridden.[14] A subclass can provide a replacement implementation that should be used whenever the receiver is a member of that class — code selection based on dynamic type — dispatch. In order to complete this dispatch, the JVM combines five pieces of information.

1. The JVM has resolved a method reference to a method with the correct name and signature, yielding a methodblock from classblock of the receiver. (Java's static typing rules enforce the restriction that the static receiver type must implement a method of the same name and signature.)
2. This statically-resolved method is inheritable. Hence there is already an entry in the static receiver class' virtual function table for the desired method. The statically resolved method has a slot in the virtual function table, and the JVM has that slot number.
3. The JVM constructed the virtual function table such that any overriding method will be in the same slot of the dynamic receiver class' virtual function table.
4. The JVM is a stack-based machine, with the arguments (beginning with the receiver) already on the stack before the `invoke` bytecode is interpreted. Therefore, given the number of arguments to the method, the JVM can peek into the stack to locate the receiver.
5. Each object (receivers can only be objects, not primitives) is ***self-describing***, meaning that it knows its dynamic type and its place in the type hierarchy. In particular, every object has a handle to its virtual function table. The structures that enable this are shown in Figures 3.12 and 3.13. Unfortunately, primitive values, integers, doubles, etc. are not self-describing — but they never appear as method receivers.

Therefore, lookup for `invoke-virtual` proceeds in six steps:

---

[14] If a class implements a private method, it can only be invoked from within that class, so any overriding implementation in a subclass could never apply. Also, Java's visibility constraints disallow a private method that would override an inheritable (`protected`, `package-private`, or `public` method).

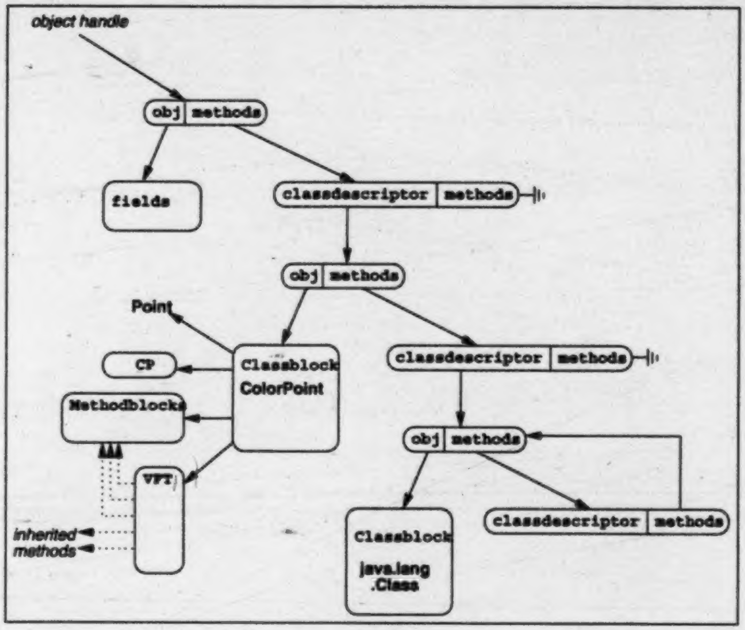

object handle
obj
methods
fields
classdescriptor
methods
obj
methods
Point
CP
Classblock
ColorPoint
Methodblocks
classdescriptor
methods
obj
methods
VFT
inherited
methods
Classblock
java.lang
.Class
classdescriptor
methods

Figure 3.12: Simplified Object and Class Structure of instance of class `ColorPoint`

1. locate the methodblock for the statically resolved method through the constant pool reference,

2. determine the slot number from the methodblock,

3. determine the number of arguments from the methodblock,

4. peek into the operand stack to locate the receiver,

5. traverse the receiver's pointer to locate the virtual function table for the receiver's dynamic type, and

6. locate the dynamically dispatched method by indexing into the given slot in the receiver's virtual function table.

All of these steps are time-consuming. Two key values are constant for a given call-site: the number of arguments and the slot number. Therefore, Sun Microsystems offers two more quick bytecodes to provide more optimization.

First, `invoke-virtual-quick`(`IVQ`) permits the slot number and argument count to replace the normal two-byte constant-pool index. This only works if the slot number is less than 256; but this is very common occurrence. In this case, lookup is reduced to three steps.

Second, `invoke-virtual-object-quick`(`IVOQ`) encodes the slot number and argument count in the same way. This bytecode differs only in that it expects the static receiver

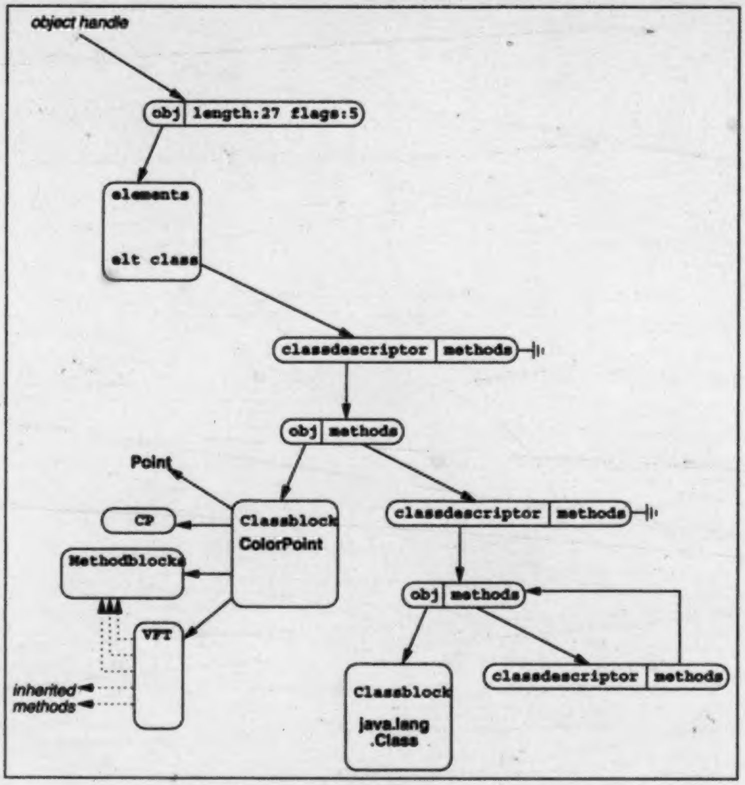

object handle
obj
length:27 flags:5
elements
elt class
classdescriptor
methods
obj
methods
Point
CP
Classblock
ColorPoint
Methodblocks
VFT
inherited
methods
classdescriptor
methods
obj
methods
Classblock
java.lang
.Class
classdescriptor
methods

Figure 3.13: Array Element and Class Structure for array of `ColorPoint`

type to be `java.lang.Object`. It is used primarily for arrays (which do not maintain their own virtual function table but use the one from `java.lang.Object`) to respond to methods such as `equals`, `hashCode`, or `clone`.

`Invoke-nonvirtual` Bytecodes

The `invoke-special`, `invoke-nonvirtual-quick`, and `invoke-super-quick` bytecodes are used exclusively to invoke private methods, constructors, and to perform "super" method calls.

In the first case, private methods, the exact class and method name and method signature are already known: the exact class must be the current class. Therefore, both of these can be directly coded into the method reference. In particular, this applies to private constructors. Therefore, no lookup is required: all private methods are *statically uni-dispatched* and precisely encoded into the method reference.

In the second case, non-private constructor calls, there are three possibilities: construction of an unrelated class, chaining to another constructor in the current class (the `this.(...)` syntax), or chaining to a superclass constructor (the `super.(...)` syntax). Again, in each of these cases, the exact class, method name, and method signature are statically known. Hence the method reference is exact, and no lookup is required.

In the third case, "super" calls to non-constructor methods,[15] the superclass is known exactly. Further, in order to compile a Java class, the compiler must have validated the current class against its super-classes, the method name and signature are known precisely too. Therefore, supercalls are ***statically uni-dispatched*** and do not require lookup.

In summary, the `invoke-special` bytecode can always be replaced with an `invoke-nonvirtual-quick` bytecode.

Sun Microsystems supplies an `invoke-super-quick` bytecode as well.[16] Sun Microsystems' JVM would only emit an `invoke-super-quick` for a super call to a non-private, non-constructor, instance method; a virtual method.[17]

The lookup-up virtual method has a two-byte slot number. Unlike the `IVQ` bytecode, the argument count is not needed to locate the receiver virtual function table, so both index bytes are available for the slot number. Therefore, in `INVSQ`, Sun Microsystems' JVM replaces the constant-pool index in the bytecode with the slot number. Method lookup proceeds as follows:

1. determine the class of the currently executing method,
2. determine its superclass,
3. locate the superclass' virtual function table,
4. index at the two-byte offset into that virtual function table.

`Invoke-interface` Bytecode

`Invoke-interface` is very similar to `invoke-virtual` except that it performs a speculative lookup first, then diverts to the same code that `invoke-virtual` used as an index into the dynamic receiver class' virtual table. Our multi-dispatch extensions will take effect at that point, giving us the full effect of multi-dispatch. An `invoke-interface-quick` recognizes that resolution is complete, and caches the speculative lookup to reduce latency.

3.3.3 Lossless Bytecodes

It is important to note that each of the quick bytecodes that we first introduced, retain direct access to the full details of the method invocation they invoke. For this reason, they are called ***lossless bytecodes***. In contrast, the quick bytecodes introduced later no longer retain the constant pool index for the desired method reference. They discard information that they no longer require. This distinction impacts the operation of JIT compilers. Unfortunately,

---

[15] Note that super can never call a private method.

[16] According to a comment in the code, this is an artifact of a bug-fix from before the `invoke-nonvirtual-quick` bytecode operated correctly.

[17] In actual fact, Sun Microsystems' JVM does not generate `INVSQ` bytecodes because the CLASS portion of the method reference already correctly names the superclass.

Sun Microsystems incorrectly classifies `invoke-super-quick` as lossless, even though it clearly does not fit that category. For a uni-dispatch Java system, this is not important because the JVM will never rewrite a bytecode to an `INVSQ`. This is a side-effect of the fact that "super" calls can be completely uni-dispatched statically. This will become an issue for Multi-Dispatch Java, because we must distinguish between `INVQ` and `INVSQ` — the former accepts the receiver type directly, the latter climbs to the receiver's superclass.

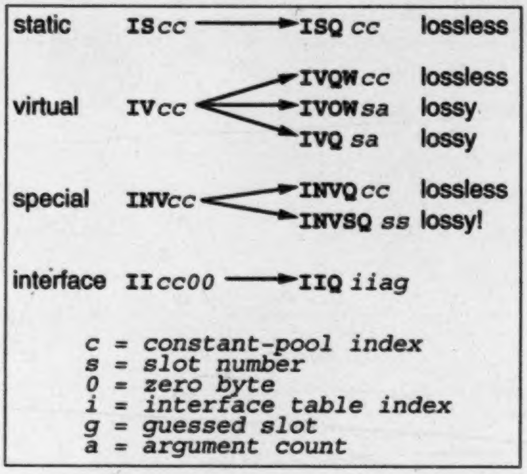

static
ISCC
ISQ cc
lossless
IVQW cc
lossless
virtual
IVcc
IVOW sa
lossy
IVQ sa
lossy
special
INVcc
INVQ cc
lossless
INVSQ ss
lossy!
interface
IIcc00
IIQ iiag
c = constant-pool index
s = slot number
0 = zero byte
i = interface table index
g = guessed slot
a = argument count

Figure 3.14: Summary of Quick Optimizations

Based upon these eleven `invoke` bytecodes, there are five different modes of invocation, as described in Section 15.12.4 of the Java Language Specification [22]:

1. *static* for static methods (including private static methods) — this applies an `IS` or `ISQ` bytecode,
2. *non-virtual* for instance constructors and private instance methods — this applies an `INVQ` bytecode (or an `INV` that proves to reference one of these two kinds of methods),
3. *virtual* for (non-private) instance methods — this applies an `IV`, `IVQ`, `IVQW`, or `IVOW` bytecode,
4. *super* for "super" calls to instance methods — this applies an `INVSQ` bytecode (or an `INV` that proves to reference an instance constructor or private method), and
5. *interface* for calls to instance methods declared as implemented for an interface — this applies an `II` or `IIQ` bytecode.

For our purposes, multi-dispatch always occurs after the lookup for an interface method invocation. Therefore, the interface mode is identical to the virtual mode, and we henceforth we treat it as such. It is valuable to note that private instance methods have yield the same result whether they are invoked as non-virtual or virtual methods. This is because they are visible only within the class that defines them. Multi-dispatch will later render this break this similarity.

3.3.4 Invocation

Once a method is resolved and looked up, a method-specific *invoker* is executed to begin the interpretation of the new method. This invoker performs method-specific operations, such as acquiring a lock in the case of `synchronized` methods, constructing a JVM activation record in the case of bytecode methods, or preparing a machine-level activation record for `native` methods. Last, the invoker transfers control to the new method.

Sun Microsystems supplies a number of custom-generated invokers optimized for frequently occurring method signatures. For example, an object receiver with zero arguments, or one, two, or three `int` arguments can be very quickly unstacked without traversing the method signature. These custom invokers take advantage of the memory layout of the JVM activation records to reduce the overhead of beginning execution of a new method.

Chapter 4

Multi-Dispatch Java — Design

We now have sufficient information to describe two general designs for extending the JVM to support multi-dispatch. We mark the methods which are to be multi-dispatched, and interpose a multi-dispatch algorithm into the interpretation of the four invoke bytecodes.

4.1 Marking Methods For Multi-Dispatch

First, we need to provide a way for designating which methods are to be multi-dispatched. Originally, we recognized three special marker interfaces which designate that all of a class' methods should be multi-dispatched. Later, we extended this to support special attributes which enable individual behaviours to be marked without including all methods in the entire class.

4.1.1 Marker Interfaces

Marking entire `.class` files without changing the language syntax is straightforward. We define three empty interfaces `VirtualMultiDispatchable`, `StaticMultiDispatchable`, `SpecialMultiDispatchable` to control application of multi-dispatch to the specific invoke bytecodes.[1]

The use of empty interfaces to control JVM operation has historical precedent in Java. For example, the `Cloneable` interface determines whether the `Object.clone()` method (which every object inherits) simply throws an exception or performs an actual cloning operation.

In particular, these are the kinds of methods affected:

- `Java/lang/VirtualMultiDispatchable` concerns all non-private, non-constructor, non-static methods with at least one object argument;

- `Java/lang/StaticMultiDispatchable` involves all non-class constructor (`<clinit>`) static methods with at least one object argument;

---

[1] Recall that `invoke-interface` performs special lookup, then joins the `invoke-virtual` codepath; hence `invoke-interface` is also controlled by the `VirtualMultiDispatchable` marker interface.

- `Java/lang/SpecialMultiDispatchable` relates to all private methods, all instance constructors `<init>`, and all non-private virtual methods with at least one object argument.

The requirement that each method must have at least one object argument recognizes that methods dispatch on their receiver, and must dispatch on another object argument before multi-dispatch has any effect.

Our interfaces are specially recognized during class loading; specifically during the Preparation phase[2], when interfaces are linked. The `.class` file retains that interface name in the constant pool and the virtual machine can easily check for each of these at class loading time. We compare each interface that a class implements against our list, and if any match, then a special flag (one for each `MultiDispatchable` interface) is set.

In accordance with the inheritance structure of Java, if one class extends another class that already has virtual multi-dispatch marked, then the subclass automatically does as well. Static and special multi-dispatch do not propagate via inheritance, because the methods they involve (static and constructor methods), do not propagate by inheritance.

Once the interfaces are prepared, the class loading operation continues to preparing methods — building the virtual function table, assigning slot numbers, and so forth. During this process, any methods that are affected by the `MultiDispatchable` interfaces have special flags set to indicate this fact. In addition, for virtual multi-methods, any inherited methodblocks are duplicated into the virtual function table. This ensures that the original definition is not marked by our special flags, but the inheriting class now has a duplicate that is marked for multi-dispatch.

This marker interface implementation does not change the syntax of the Java programming language or the binary `.class` file format in any way.

Our interface-based technique allows us to retain compatibility with existing programs, compilers, and libraries. Any class that implements our marker interface has different semantics for dispatch. However the semantics of existing uni-dispatch programs and libraries are not changed since they do not implement the interface. The programmer retains complete control and responsibility for designating multi-dispatchable classes. This allows the developer to consciously target the multi-dispatch technique to known programming situations, such as double dispatch.

4.1.2 Multi-Dispatchable Attributes

We observed that the propagation of the marker interface flags to the methods themselves could easily be extended to allow individual methods to be marked as multi-dispatchable. Therefore, a simple addition allows any methods with a new attribute with name `Java/lang/VirtualMultiDispatchable`, `Java/lang/StaticMultiDispatchable`, or `Java/lang/SpecialMultiDispatchable` to be marked for multi-dispatch. For consistency, this marker automatically includes any methods with the same name, arity, and corresponding arguments (object in argument positions where the marked method has objects, primitive arguments must be identical in type and position).

---

[2] Refer to Section 5.4.3.4 of the JVM Specification [32]

The JVM specification, Section 4.7.1, states that

> Java virtual machine implementations are permitted to recognize and use new attributes found in the `attributes`tables of `class`file structures. However any attribute not defined as part of this Java virtual machine specification must not affect the semantics of `class` or `interface` types. Java virtual machine implementations are required to silently ignore attributes they do not recognize.

Therefore, we can include these new attributes without generating non-conforming Java.

This attribute-based extension focuses multi-dispatch more finely than the entire class approach of the marker interfaces. However it requires separate tool support, whereas the marker interfaces work with any standard Java compiler. Other than simple validations, we do not exploit attribute-based multi-dispatch markers at this time. Specifically, we supply neither a tool to insert these custom attributes into a pre-existing `.class` file, nor a compiler that emits these custom attributes. Future projects might consider this avenue of research.

4.2 Locating the Multi-Dispatcher

Now that we have informed the JVM about which methods require multi-dispatching, it can proceed to effect multi-dispatch. On the face of it, the process is simple: whenever a method is about to be invoked, check whether it should be multi-dispatched, and if so, find the method that most precisely matches the types of the arguments – dynamic multi-dispatch.

We can interpose this multi-dispatch operation in either of the last two phases of interpreting an invoke bytecode: lookup or invocation.

The conservative approach is to replace the invoker for multi-methods with one that selects a more specific method based on the actual arguments. Hence, existing uni-dispatch method invocations are unchanged in any way. However the multi-invoker needs to know the specific `invoke` bytecode involved. This is because a virtual method might be called via "super", or directly. The multi-method lookup differs between these two — a "super" must use the superclass of the current host class to determine the multi-method, whereas a direct invocation must not.

At dispatch time, our multi-invoker executes instead of the original JVM invoker. Our invoker locates a more-precise method based on the dynamic types of the invocation arguments and executes it in place of the original method.

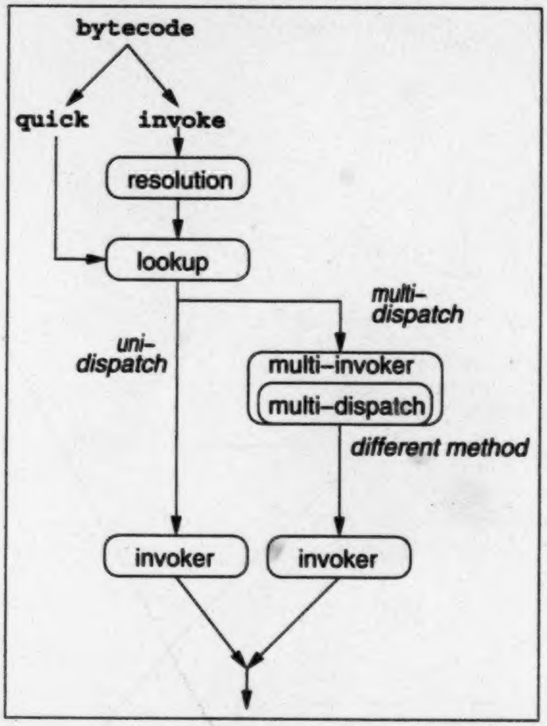

bytecode
quick
invoke
resolution
lookup
multi–dispatch
uni–dispatch
multi–invoker
multi–dispatch
different method
invoker
invoker

Figure 4.1: Multi-Invoker Based Multi-Dispatch

The aggressive approach is to inline the multi-method lookup as part of the normal lookup. This adds additional machine instructions to the uni-dispatch code-path, but considerably simplifies the multi-dispatch operation — in particular, we no longer need to reconstruct the `invoke` bytecode. An additional performance obstacle is that method lookup is done within the core interpreter loop, which is written in assembler (although a debugging version of the loop is available in C).

Either approach will ensure that whenever the JVM encounters an invocation of a method marked for multi-dispatch, the multi-dispatcher has an opportunity to select the most specific method — the subject of our next section.

4.3 The Multi-Dispatcher

Now we turn our attention to the high-level function of the multi-dispatcher itself. Structurally, the multi-dispatcher comprises four distinct dispatch routines, one for each method invocation mode: static, non-virtual, virtual, and super. The mode is determined by the invocation bytecode, and serves to limit the set of methods considered as potential multi-

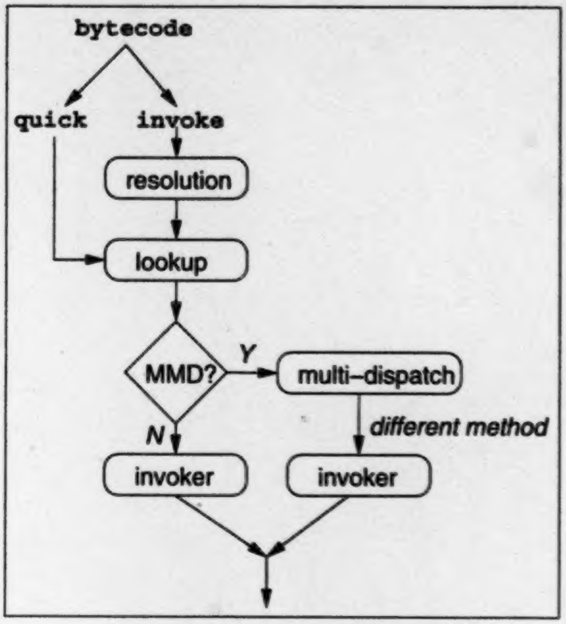


Figure 4.2: Inlined Multi-Dispatch

dispatch targets.

The multi-dispatcher operates in a straightforward fashion. It first determines the invocation mode — directly in the inline case, or by reconstructing the `invoke` bytecode from the current activation record. It then calls a routine customized for that invocation mode. Each of those routines follow a two-step strategy: determine the types of the receiver and other arguments by examining the uni-dispatched method and walking the argument stack, and select the more-precise method to invoke.

The operation of the multi-dispatcher depends directly on the different invocation modes, and their semantics. We look at them next.

4.3.1 Multi-Dispatch Semantics in Java

Consider a call-site where the JVM is about to invoke a method that is marked for multi-dispatch. The basic tenet of multi-dispatch is that *the method invoked is the most specific applicable to the types of the arguments at hand.* For static multi-dispatch, the types are given by type annotations or type inference. In the case of dynamic multi-dispatch, these types are derived from the actual argument values. However the question remains: what characterizes this "most specific applicable" method?

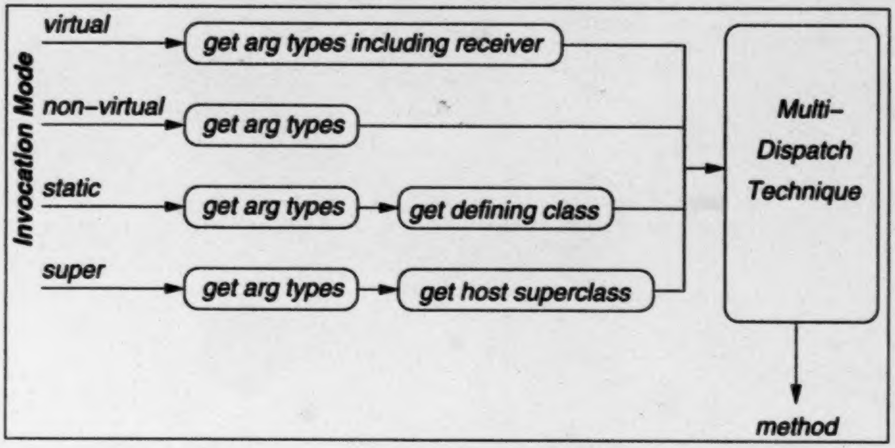


Figure 4.3: Structure of the Multi-Dispatcher

Accessibility

The first aspect in deciding which method to invoke is ***accessibility***. This is determining whether a method is visible based on the current execution context. For example, private methods can only be accessed from within their defining class, package methods must be invoked by a member of the package, and so forth. The Java Language Specification [32] specifically notes that method selection for static multi-dispatch is to be guided by accessibility. For `Javac`, if a method is not accessible, then it does not apply.

Alternatively, the Java Virtual Machine is responsible for enforcing access controls, not applying them. For example, if method resolution located a private method in another class, then the JVM would not discard that private method and continue searching for a more visible one. The JVM would throw an `IllegalAccessError`. We choose to have dynamic multi-dispatch operate in the same way — it ignores accessibility when determining a multi-dispatch method. If the found method is inaccessible, then the default behaviour, throwing an `IllegalAccessError`, is followed. A different tool, `MDLint`, can provide off-line checks for inaccessible multi-methods, potentially ambiguous invocations, visibility reductions by overriding methods, and so forth.

In addition, the JVM is responsible neither for ensuring that all thrown exceptions are listed as part of `throws` clause, nor that overriding method propagate the `throws` clauses. Although the `throws` information is available in the `.class` file as part of the `Exceptions` attribute for each method, the JVM is not required to ensure that any thrown exception is listed in that attribute. This is explicitly declared in Section 4.7.4 of the JVM Specification [32]. This property must be separately checked, by a separate tool such as `MDLint`.

Applicable

We begin by defining *applicable*, then proceed to discuss *specificity*. In order to have dynamic multi-dispatch provide similar semantics as the static multi-dispatch given for Java, we follow the same development given in Section 15.12.2 of the Java Language Specification [22].

The intuitive definition is that a method applies if it has the correct name and number of arguments, and that each parameter type it accepts is a super-type of the corresponding argument. More formally, we begin by defining applicability in terms of type specificity.

Definition 1 (More Specific — Type) *A type* $\mathtt{T}_1$ *is more specific than another type* $\mathtt{T}_2$ *(denoted as* $\mathtt{T}_1 \preceq \mathtt{T}_2$*) iff* $\mathtt{T}_1$ *is a subtype of* $\mathtt{T}_2$.

C++ uses the term *dominates* for this relationship [44]. It is clear the definition of more specific is reflexive, transitive, and anti-symmetric. It imposes a partial order on the hierarchy of types.

Definition 2 (Applicable) *Given a method invocation* $\mathtt{R.name}_1\mathtt{(A}_1\mathtt{,\ A}_2\mathtt{,\ \ldots,\ A}_n\mathtt{)}$*, with receiver type* `R`*, selector* $\mathtt{name}_1$*, and argument types* $\mathtt{A}_1\mathtt{,\ \ldots,\ A}_n$*, a method implementation* $\mathtt{S.name}_2\mathtt{(P}_1\mathtt{,\ P}_2\mathtt{,\ \ldots,\ P}_m\mathtt{)}$ *in class* `S` *with selector* $\mathtt{name}_2$ *and parameter types* $\mathtt{P}_1\mathtt{,\ \ldots,\ P}_m$ *applies at the invocation iff*

1. $\mathtt{name}_1 = \mathtt{name}_2$,
2. `S` *is a (non-strict) subtype of* `R` *(denoted* $\mathtt{S} \preceq \mathtt{R}$*),*
3. $m = n$ *(the number of arguments equals the number or parameters),*
4. $\mathtt{P}_i \preceq \mathtt{A}_i \quad i = 1 \ldots m$.

Because method types form a product type over the argument types, the notion of applicability transforms a simple type hierarchy into a lattice of method implementation types. For example, consider a simple type hierarchy consisting of a superclass `Super` and a subclass `Sub`. Java automatically includes two other types: `Object` and `null`. Then, a binary method on this simple hierarchy forms the lattice given in Figure 4.4. The type lattice permits us to determine that given arguments `Super` and `Sub`, method implementations defined for `SuperxSub`,`SuperxSuper`, `SuperxObject`, `ObjectxSub`,`ObjectxSuper`, and `ObjectxObject` all apply.

More Specific

Clearly, the `SuperxSub` implementation is preferable — it most closely matches the actual argument types. We formalize that notion of more specific here.

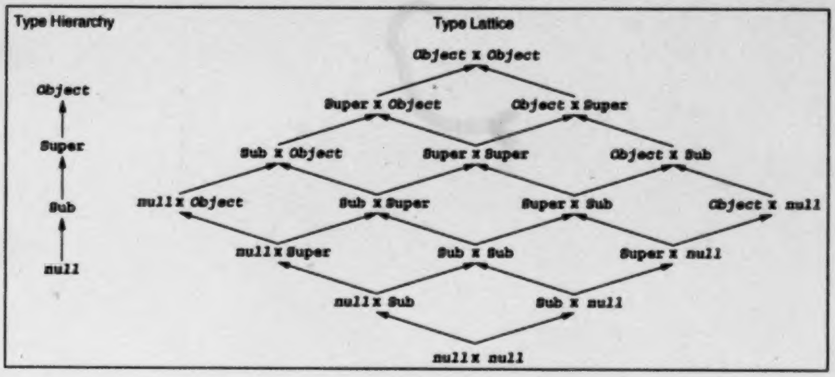

Type Hierarchy
Object
Super
Sub
null
Type Lattice
Object x Object
Super x Object
Object x Super
Sub x Object
Super x Super
Object x Sub
null x Object
Sub x Super
Super x Sub
Object x null
null x Super
Sub x Sub
Super x null
null x Sub
Sub x null
null x null

Figure 4.4: Type Hierarchy Induces Method Type Lattice

Definition 3 (More Specific — Method) *For a given method invocation, and given two method implementations applicable to that invocation,* $M_1 =$`S₁.name(`$P_{1,1}$`, `$P_{1,2}$`, ..., `$P_{1,m}$`)` *and* $M_2 =$`S₂.name(`$P_{2,1}$`, `$P_{2,2}$`, ..., `$P_{2,m}$`)`*,* M_1 *is more specific than* M_2 *iff*

1. $S_1 \preceq S_2$, *and*

2. $P_{1,i} \preceq P_{2,i}, \quad i = 1 \ldots m.$

Indeed, this definition is exactly that given by the standard extension of types by Cartesian product. It is also reflexive, transitive, and anti-symmetric. It imposes a partial order on method implementations.

This partial order is the multi-dispatch equivalent of *method overriding*. In uni-dispatch Java, methods are overridden only on the receiver class, giving an ordering similar to that of the type hierarchy in Figure 4.4. For example, a method `Super.method(Super s)` is uni-dispatch overridden by `Sub.method(Super s)` — they both accept the identical arguments — both in number and type. Although `Super.method(Sub s)` overloads the definition of `Super.method(Super s`, it is examined only during static multi-dispatch.

For dynamic multi-dispatch, the receiver and all of the arguments are considered at execution time. Therefore, the receiver and all of the argument types are involved in the definition of overriding. For example, `Super.method(Super s)` is multi-dispatch overridden by both `Super.method(Sub s)` and `Sub.method(Super s)`. The partial order given by the more-specific relation is the basis for saying one method overrides another.

Maximally Specific

With this understanding of overriding, we can proceed to find the maximal overriding method — the one that is most specific to the arguments at hand.

Definition 4 (Maximally Specific — Method) *For a given method invocation, and a* *set of method implementations* $\{M_i, \quad i = 1 \dots j\}$ *applicable to that invocation, a method implementation* M_x *is maximally specific iff* $M_i \not\preceq M_x, \quad i = 1 \dots j, x \neq i$.

It is easily observed that every finite, non-empty set of method implementations contains a maximally specific implementation. However it is important to realize that a set of method implementations may not contain a unique maximally specific meth[illegible]. Consider the example lattice above, with a method invocation with types `SuperxSub`, but only the following methods implemented: { `SuperxSuper`, `ObjectxSub`, `ObjectxObject` }. Clearly, all of the implemented methods are applicable. The first implementation is more specific than the third, and likewise, the second implementation is more specific than the third. However neither of the first two implementations is more specific than the other. Hence, this set has two maximally specific implementations.

Most Specific

In the case that a unique maximally specific implementation exists, we call it the most specific implementation. It is easily observed that this uniqueness ensures that the most specific implementation is more specific than every other implementation.

Definition 5 (Most Specific — Method) *For a given method invocation, and a set of method implementations* $\{M_i, \quad i = 1 \dots j\}$ *applicable to that invocation, a method implementation* M_x *is most specific iff it is a unique maximally specific implementation, or equivalently,* $M_x \preceq M_i, \quad i = 1 \dots j$.

The goal of multi-dispatch is to determine this most specific applicable method implementation for each call-site. This determination depends upon the specific `invoke` bytecode at the call-site. We review the four different modes of method invocation, and determine the set of method implementations used in computing the most specific applicable implementation.

4.3.2 `Invoke-static` Multi-Dispatch

As we saw previously (see Section 3.3.2), the `invoke-static` bytecode uses a statically known "pseudo-receiver" class. In essence, the receiver class name is used to provide a simple name-space for procedures. To preserve this semantics, we restrict the implementations considered for applicability by our multi-dispatcher to those static methods defined within that "pseudo-receiver" class.

Therefore, for a method invocation of `R.name(...)` by an `invoke-static` (`IS` or `ISQ`) bytecode, the multi-dispatcher will select the most specific applicable method from the set:

{ *M* | *M* is static, *M* is defined in class *R* }

4.3.3 Invoke-nonvirtual Multi-Dispatch

As we saw previously, the `invoke-special` bytecode subsumes three different kinds of invocations, all of which are statically known, and hence not "virtual" for uni-dispatch. They are:

1. private non-static methods which have their receiver class statically known — it must be the class of the currently executing method,
2. constructors which also have the specific receiver class statically known — it is the class that is being constructed,
3. "super" calls to a virtual method — these know the current executing class, and must dispatch to another method in a superclass.

It is important to note that in the first two cases, the list of potential method implementations for dispatch is limited to those within a single class. In the former case, this is because private methods are not accessible outside of their declaring class. In the latter case, this is because constructors are not inherited; hence the receiver type of a "super"-constructor call is certain. Internally, the JVM treats `invoke-super` differently, and we examine it separately in the next section. Multi-dispatch Java agrees with uni-dispatch Java in the analysis of the first two cases, hence the class given in a method reference for `invoke-nonvirtual` is correct.

Therefore, for a method invocation of `R.name(...)` by an `invoke-nonvirtual (INVQ)` bytecode, the multi-dispatcher will select the most specific applicable method from the set:

$$\{\ M \mid M \text{ is not static, } M \text{ is defined in class } R\ \}$$

The "not static" requirement is not superfluous. Although `Javac` does not permit uni-dispatched static and virtual methods to override each other, it enforces no such limitation on multi-dispatch overriding. For example, the class definitions in Figure 4.5 are correct in Java. The class `Super` contains two methods, one virtual and one static, that have matching selector names and argument lists.

```
class Super {
  void method(Super s) {
    System.out.println("Super.method(Super)");
  }
  void static method(Sub s) {
    System.out.println("Super.method(Sub)");
  }
}
class Sub extends Super { ... }
```

Figure 4.5: Static and Virtual Methods Interfere

4.3.4 Invoke-super Multi-Dispatch

The `invoke-special` bytecode is used to call methods in a third way. That third case, a "super" call, is not so simple. The receiver class found in the method is statically known *for uni-dispatch* and can be encoded exactly. However multi-dispatch cannot depend upon this uni-dispatch receiver class. Let us see why.

For a "super" call, `Javac` has statically multi-dispatched to a virtual method somewhere in the currently-executing method's superclass chain. That method might be in the immediate superclass, but it could have been located several levels up in the superclass chain. We see an example of this in Figure 4.6.

Uni-dispatch Java ignores the `B.method(Sub)` implementation, because the static type of the argument is `Super`. Hence, the `invoke-special` bytecode references the method `A.method(Super)`. A uni-dispatch JVM accepts that the method required is in class `A` — and optimizes the bytecode to an `INVQ` that directly references that `A.method(Super)`. The uni-dispatch JVM turns the "super" call into a direct, non-virtual method invocation.

```
// a hierarchy of argument types
class Super { ... }
class Sub { ... }
class A {
  void method(Super s) {
    System.out.println("A.method(Super)");
  }
}
class B extends A {
  void method(Sub s) {
    System.out.println("B.method(Sub)");
  }
}
class C extends A {
  static public void main(String args[]) {
    C c = new C();
    Super s = new Sub();
    c.method(s);
  }
  void method(Super s) {
    super.method(s);
  }
}
```

Figure 4.6: Super Invocations Skip Levels

Multi-dispatch Java cannot rely upon the accuracy of that direct `INVQ` bytecode. Hence, we ensure that the more accurate, (but more expensive) `invoke-super-quick` (`INVSQ`) byte code is emitted — ordinarily the JVM does not generate that bytecode.[3] By applying this bytecode, the multi-dispatch JVM will recognize when a super occurs, and be able to consider all of the applicable methods.

---

[3] Since that bytecode was never emitted, several bugs were discovered in its implementation, and the OpenJIT support.

Therefore, for a method invocation of `super.name(...)` within a currently executing method `H.m(...)` by an `invoke-super-quick` (`INVSQ`) bytecode, the multi-dispatcher will select the most specific applicable method from the set:

{ `M` | `M` is not static, `M` is defined in or inherited by the superclass of `H` }

Finally, we can decide the multi-dispatch target of the original bytecode, `invoke-special`. First, we must decide what kind of invocation mode it is expected to perform: a private method or constructor multi-dispatch just like their optimized version `INVQ`, and a "super" call multi-dispatches just as its optimized, `INVSQ`, does.

4.3.5 Invoke-virtual Multi-Dispatch

For `invoke-virtual` calls, we must consider methods that may come from any level in the inheritance chain. Fortunately, the uni-dispatch operation has already found the virtual function table for the exact dynamic class of the receiver. That virtual function table contains many of the methods that could be used for multi-dispatched.

However private methods prove to be a quandary. Consider the code in Figure 4.7. In both of the method invocations in the `main` procedure, the receiver type is `Super`, and the argument type is `Sub`. In the first instance, the compiler has enough information to static multi-dispatch accurately — but it uses a special bytecode, `INV`, to reflect the private visibility of the target method. In the second case, the compiler inaccurately multi-dispatches to `Super.method(Super)`, and emits an `invoke-virtual` bytecode.

```
class Super {
  void method(Super s) {
    System.out.println("Super.method(Super)");
  }
  private void method(Sub s) {
    System.out.println("Super.method(Sub)");
  }
  static public void main(String args[]) {
    Super super = new Super();
    Sub sub1 = new Sub();
    Super sub2 = new Sub();
    super.method(sub1); // invoke-special of private!
    super.method(sub2); // invoke-virtual of private?
  }
}
class Sub extends Super { ... }
```

Figure 4.7: Private Method Targeted By Multi-Dispatch

We choose to include the private method in the collection of potential multi-dispatch targets, for three reasons. First, including the private method provides greater referential transparency — a key benefit of multi-dispatch. In particular, consider the following problematic invocation: `super.method(new Sub());`.

In this case, the private method would have been invoked because the compiler would have the precise type information at compile time. We wish to invoke the same method that the compiler would have selected given exact type information. Second, the JVM specification [32] for `invoke-virtual` does not expressly forbid invoking a private method through it. Third, we expect that multi-dispatch via `invoke-special` might also locate non-private methods. Therefore, we treat invocation of private (non-static, non-constructor) multi-methods just like `invoke-virtual`.

Indeed, it would appear that applying `invoke-special` for private methods is primarily a performance optimization. Uni-dispatch can avoid any lookup — the receiver is statically known and the arguments are statically dispatched.

Therefore, for a method invocation `R.name(...)` with dynamic receiver type `S` by an `invoke-virtual` (`IV`, `IVQ`, `IVQW`, `IVOW`) bytecode, the multi-dispatcher will select the most specific applicable method from the set:

{ `M` | `M` is not static, `M` is defined in or inherited by class `S` }

4.4 Potential Errors

Although the arguments are compatible, there are still three potential errors that may arise when the method found by multi-dispatch might differ from the uni-dispatch one.

4.4.1 Ambiguous Invocations

First, there may not be a unique maximally specific implementation. For example, consider the type lattice from Figure 4.4, with implementations defined for `Super.method(Sub)` and `Sub.method(Super)` as shown in Figure 3.2(a). Given an invocation with receiver and argument of type `Super`, both implementations apply and neither method is more specific than the other. Both are maximally specific.

If the static types allow the compiler to recognize the ambiguity, `Javac` emits an error message as described in the Java Language Specification [22], Section 15.12.2.3. It is possible for the static types to demonstrate no conflict, but the dynamic types still generate an ambiguity. At execution time, the multi-dispatch JVM must report the problem, and the accepted technique is to throw an exception. Therefore, we declare a new exception, `java.lang.AmbiguousMethodError`, a subclass of `IncompatibleRuntimeError` that the virtual machine will throw when presented with an ambiguous invocation. As a debugging aid, all maximally specific methods are available as a field of the thrown exception and can be examined using Java's reflection API.

The solution to avoid ambiguous invocations is to define a *conflict method* that provides a most specific applicable method implementation for each ambiguous pair.[4] It is easily shown that the conflict implementation must be defined as accepting, for each parameter position (including the receiver), the more specific of the two parameter types. For instance, the conflict method for the example given must accept `Sub` as the receiver and the first argument. We see this in Figure 4.8(b).

---

[4] This conflict method cannot introduce the need for additional conflict methods.

```java
class Super {
 void method(Sub s) {
  System.out.println("Super.method(Sub)");
 }
}
class Sub extends Super {
 void method(Super s) {
  System.out.println("Sub.method(Super)");
 }
}



class Main {
 static public void main(String args[]) {
  Super super = new Super();
  Sub sub1 = new Sub();
  Super sub2 = new Sub();
  sub1.method(sub1); // compiler -> ambiguous!
  sub2.method(sub2); // md runtime -> ambiguous!
 }
}
```

(a) Ambiguous Methods

```java
class Super {
 void method(Sub s) {
  System.out.println("Super.method(Sub)");
 }
}
class Sub extends Super {
 void method(Super s) {
  System.out.println("Sub.method(Super)");
 }
 void method(Sub s) { // conflict method
  System.out.println("Sub.method(Sub s)");
 }
}
class Main {
 static public void main(String args[]) {
  Super super = new Super();
  Sub sub1 = new Sub();
  Super sub2 = new Sub();
  sub1.method(sub1); // compiler -> okay!
  sub2.method(sub2); // md runtime -> okay!
 }
}
```

(b) Conflict Method Defined

Figure 4.8: Ambiguous Dispatch and Conflict Method

4.4.2 Incompatible Return Types

The second potential error which the multi-dispatch JVM may encounter is an incompatible return type. Standard Java programs are only checked to ensure that uni-dispatch overriding methods have `no-variant` return types. This means that a subclass which declares a uni-dispatch overriding method must have exactly the same return type in the new implementation. This validation ensures that the use of return values is type-safe.

However the definition of uni-dispatch overriding does not recognize the larger set of methods which multi-dispatch overriding does. Therefore, the set of methods verified for compatible return types by `Javac` does not include all methods that might be considered by the multi-dispatcher.

For example, consider the program found in Figure 4.9. Standard Java accepts this program, because the only uni-dispatch override for `Super.method(Super)` is `Sub.method(Super)`, the uni-dispatch override. Standard Java does not consider `Super.method(Sub)` as applicable to the method invocation in `main`. Therefore, its return type, `int`, is not checked as acceptable for assignment to a `String` — it is not a potential overriding implementation of `Super.method(Super)`. However multi-dispatch Java will, in this example, select `Super.method(Sub)` as the implementation to invoke. The assignment of an `int` value to a `String` is not type-safe and cannot be permitted.

Hence, although standard Java permits overloaded methods to return different argument types, multi-dispatch Java must not in order to preserve type-safety. Furthermore, the Java Language Specification explicitly notes that return types are *not* to be used to remove methods from consideration during static multi-dispatch (see Section 15.12.2.4). Instead, the compiler is obligated to emit an error message and abort the compilation. In keeping with this design, the multi-dispatch JVM must also report an error when an incompatible return type is discovered. For this purpose, another new exception has been defined: `java.lang.IllegalReturnTypeError`, also a subclass of `IllegalClassChangeError`.

```
class Super {
  String method(Super s) {
    return "Super.method(Super)";
  }
  int method(Sub s) {
    return 1;
  }
}
class Sub extends Super {
  // return type constrained
  String method(Super s) {
    return "Super.method(Super)";
  }
}
class Main {
  static public void main(String args[]) {
    Super super = new Super();
    Super sub = new Sub();
    String s = super.method(sub); // md - illegal return type!
  }
}
```

Figure 4.9: Illegal Return Type Error

Relaxing Return Type Restrictions

Java has adopted a stringent return type matching regime. Return types of overriding methods must be `no-variant` — that is, the overriding method must return a value of identical type to that of the overridden method. C++, prior to the third edition [46] noted that "it is an error for a derived class function to differ from a base class' virtual function in the return type only." This means that an overriding method — one with argument types identical to an inherited method, must have an identical return type also. However this restriction has been shown to be stronger than needed [38].

Hence, in C++ [46], this restriction was relaxed to permit *co-variant* [8] return types. Overriding methods are now permitted to return a subtype of the original method's return type. It is clear that this does not compromise the type-safety of the programming language, because the value returned by the overriding method is still compatible with the super return type. Using the previous type hierarchy, if `Super.method(Super)` returns a `Super`, and `Sub.method(Super)` returns a `Sub`, the current Java compilers report an error — the overriding method return type (`Sub`) doesn't match the original (`Super`). However clearly, an instance of `Sub` is also a `Super` by the substitutability principle. Unfortunately, we cannot stop the compiler from generating an error in this situation. Otherwise, we could dispatch this case.

Sun Microsystems agrees with this argument, and has incorporated co-variant return types into their Java extension for parametric types [48].

4.4.3 Accessibility

First, the most specific applicable method implementation may not be accessible from the call-site. For example, the most specific applicable method may have package visibility, but be invoked from outside of the package. Or, a private method might be invoked from outside its declaring class. In each of these cases, the JVM must enforce security. So, as with any method invocation, an `IllegalAccessError` can be thrown to reflect this condition. Indeed, the normal operation of the interpreter already provides this facility — the error is detected and reported at method invocation time.

4.4.4 Reporting Multi-Dispatch Errors

In each of the three situations outlined above, the multi-dispatch JVM must throw an exception to report the illegal condition. However the error can be recognized in at least two different places during execution: at class-loading time and at method invocation time.

First, we discuss class-load time error recognition. Potentially ambiguous multi-methods can be identified at class load time by recognizing that the conflict method is not implemented. It is also possible to recognize a potential illegal access error at class load time. For instance, it might be caused by a private method which is more specific (hence multi-dispatch overrides) than another method with public visibility. A potential illegal return type can be recognized by checking return types at class load time — any method that is more specific than another must have co-variant return type.

ByteCode Verification

Each of these tasks is an extension of the bytecode verifier described in the JVM Specification, Section 4.9 [32]. That description gives a four-pass process to validate a classfile and the bytecode it includes. Pass 1 provides simple consistency checks: are magic numbers valid, are attributes the correct length, does the constant pool contain no "superficially unrecognizable information", etc. Pass 2 checks static conditions: final classes cannot be subclasses, every class (except `Object`) has a superclass, methods and fields have valid names, classes, and type descriptors. However Pass 2 "does not check to make sure that the given field or method actually exists in the given class, nor does it check that the type descriptors given refer to real classes." This is postponed until Passes 3 and 4 explicitly for efficiency reasons — specifically classes and methods that are never referenced will not be checked. Pass 3 is responsible for the bulk of the verification: it validates a number of conditions on methods and invocations to ensure the integrity of the JVM. However many of its tests "are delayed until the first time the code for the method is actually invoked. In so doing, Pass 3 of the verifier avoids loading class files unless it has to." Pass 4 completes these tests at method invocation time.

Unfortunately, each of the problematic situations with multi-dispatch requires that the details of argument types and return types must be known. To check these problems at class-loading time would force immediate loading of all referenced classes, contradicting the "lazy" class-loading nature of the Java Virtual Machine.

The second option, checking at method invocation, is preferable. Clearly, these same checks can be executed during Pass 4 (at method invocation) as well — the tests occur during the method selection process. This is our design for Multi-Dispatch Java. As noted in the JVM Specification, "errors that are detected in Pass 4 cause instances of subclasses of `LinkageError` to be thrown." Hence, our errors are subclasses of this generic error.

With this high-level design, we proceed to describe our implementations.

Chapter 5

Multi-Dispatch Java — Implementation

Our implementations consist of two parts: a multi-dispatch placement, and the multi-dispatch technique. We implemented two different multi-dispatch placements: a multi-invoker and an inline multi-dispatch tester. We implemented two different dispatch algorithms. First, MSA implements a dynamic version of the Java Most Specific Applicable algorithm used by the `Javac` compiler. Second, Single Receiver Projections (SRP) [25, 24, 26] is a high performance table-based technique developed at the University of Alberta.

5.1 Multi-Dispatch Placement

We implemented two different ways to interpose multi-dispatch into the method invocation process. The first is to create a new method invoker, called the *multi-invoker* which selects an alternate, more specific method to invoke. The other was to test the multi-dispatch markers when the `invoke` bytecode is being interpreted. This second approach puts the test for multi-dispatch in line with the interpreter loop.

5.1.1 Multi-Invoker

The multi-invoker offers the benefit of removing the overhead of multi-dispatch from the uni-dispatch code-path. Other than differences in compiler optimization and memory/cache effects, uni-dispatch code executes exactly the same sequence of instructions in the interpreter loop as the original does.

As part of the uni-dispatch of an `invoke` bytecode, the JVM finds a method pointer from the array of methods in the receiver argument class. At this point, the interpreter loop is about to build a new frame to execute the found method. The interpreter loop (and classic VM JIT compilers) proceed to call a special function, called the `invoker` that handles the details of building the new frame and starting the new method. The Research JVM uses different invokers for native, bytecode, synchronized, JIT-compiled, and other method types. Similar to the OpenJIT system [35], we replace this invoker function with a custom *multi-invoker* that computes the correct multi-dispatch method. Once the more precise method is known, we simply invoke it directly.

The multi-invoker is installed at class-load time. Every multi-method has its invoker replaced with the multi-invoker, and the original invoker is cached into the methodblock. For inherited methods, we cannot simply replace the invoker. If the declaring class is not marked for multi-dispatch, then invocations of its methods with the uni-dispatch receiver should not be multi-dispatched. Therefore, methodblocks inherited from uni-dispatch classes are duplicated into fresh methodblocks that have their flags and invokers changed to support multi-dispatch.

Using the multi-invoker, the interpreter loop and invoker for uni-dispatch are unchanged. This supports our claim that uni-dispatch programs and libraries suffer no execution time penalties.

5.1.2 Inline Dispatch

The penalty for keeping the multi-dispatch overhead out of the uni-dispatch code-path is that the interpreter loop (and JIT'ed code) must recover the actual invocation bytecode from the current activation record. This can only be done by reconstructing the current execution frame in order to determine the invocation mode. In the interpreter loop, that information is directly accessible. At a cost of 3 machine instructions in the interpreter loop[1], the interpreter can divert execution into the correct multi-dispatch routine immediately. Careful placement of data can ensure that the appropriate flag values are available in cache, resulting in a negligible overhead in the uni-dispatch operation.

Within the OpenJIT compiler, there are routines, `OpenJIT_invoke_virtual`, `OpenJIT_invoke_special`, `OpenJIT_invoke_static`, and `OpenJIT_invoke_super`, corresponding to the interpreter code for each invoke bytecode. These routines perform the resolution and lookup phases of interpretation. Bytecode involving the invoke bytecodes get converted into jumps to these routines. At that point, the JVM can test the appropriate multi-dispatch flag and divert to the appropriate `Select-*-MultiMethod` routine.

5.2 Dispatch Techniques

We have experimented with two different multi-dispatch techniques; they are examined in the following sections.

---

[1] See Figure D.2

5.2.1 Reference Implementation: *MSA*

Our reference implementation is an extension of the Most Specific Applicable algorithm described in section 15.11 of *The Java Language Specification* and in section 3.2 of this dissertation. In particular, we re-examine the steps described in section 3.2 in light of the dynamic argument types being used.

When the multi-invoker is called, it has access to the methodblock that has already been found by the uni-dispatch resolution mechanism. It also has the address of the top of the operand stack, so it can peek at each of the arguments. Last, the multi-invoker has the actual receiver, which can provide the list of methods (including inherited ones) that the receiver implements.

Every method is represented by a methodblock containing many useful pieces of information. First, it holds the name of the method. Second, it contains a handle to the class that contains this method.[2] Third, it contains the signature which can be parsed to get the arity and type names of the dispatchable arguments. For performance, our implementation parses the signature only once. We add two fields to the methodblock: `int arity` to cache the arity and `ClassClass **argClass` to hold the class handles for the dispatchable arguments.

With these three pieces of information, we implement a dynamic version of the MSA algorithm directly. Wherever the original algorithm would use the static type of an argument, the JVM applies the known dynamic type instead. In the original MSA algorithm, the compiler would compare the static type of each argument with the corresponding declared type for the candidate method. In the dynamic case, the interpreter has the arguments on the stack, so it can find their dynamic types. Each argument's dynamic type is compared against the declared type of the corresponding argument of the method. Any method that is not applicable due to access rights (`private` methods) or whose declared types do not match the arguments on the stack is discarded. The remaining methods are *dynamically applicable*.

The issue of null-valued arguments becomes significant at this point. JLS chapter 4 recognizes the need for a *null type* to represent (untyped) null values. It further declares in section 4.1 that the null type can be coerced to any non-primitive type. Also, section 5.1.4 allows null types to be widened to any object, array or interface type. Statically, this means that an (untyped) null argument can be widened to any class. In the dynamic case, we want to do the same. Therefore, whenever the interpreter encounters a null argument we accept the conversion of that null to a method argument of type class, array, or interface.

Unfortunately, if the method invocation includes a null argument, the culling process retains methods which accepts arguments of classes that are not yet loaded. The multi-dispatcher needs to force these classes to be loaded to ensure that the next step operates correctly.

---

[2] Recall that methods might be inherited; this class handle is the original implementing class.

Given the list of applicable methods, the MSA algorithm finds the unique most specific method. Again the operation is identical to the process that the `Javac` compiler follows. One applicable method is tentatively selected as the most specific. Each other applicable method is tested by comparing argument by argument (including the receiver argument) against the tentatively most specific. At each step, the multi-dispatcher discards any methods that are less specific. It continues this process until only one candidate method remains, or two or more equally specific methods remain. In the latter case, the invocation is ambiguous and the multi-dispatcher throws an `AmbiguousMethodException` to advertise this fact.

Last, the multi-dispatcher verifies that the return type for the more specific method is compatible with the compiler-selected method. This check relaxes JLS 8.4.6.3, where any invocation that has a different return type must be rejected. Even so, our relaxation still maintains type-safety. If the return type is incompatible, we throw an `IllegalReturnTypeError` at runtime.

Multi-Threading Support

The MSA algorithm implicitly supports multi-threaded applications, because it accesses shared information — the classblock and methodblocks — in read-only fashion. The contents of the Java stack are thread-local, and all other values are local variables. Therefore, MSA incurs no performance penalty in multi-threaded applications.

5.2.2 Tuned SRP Dispatcher

Our second implementation provides high-efficiency dispatch in a production Java Virtual Machine, supporting the full language semantics, including interface types, `null` arguments, and array sub-typing. It accurately dispatches multi-methods while retaining the "lazy" class-loading properties of Java.

Our dispatcher comprises four parts: behaviour management, type-numbering, implementation registration, and the actual dispatch algorithm itself.

Behaviour Management

The data structure which allows multi-methods to be efficiently dispatched is called a *behaviour*. It is an equivalence class of multi-method implementations, and the associated information to enable efficient method dispatch.

Recalling the definition of multi-method overriding from Section 4.3.1, we recognize that three values uniquely identify a behaviour: a declaring type, a selector, and a terse signature.

The first value, a declaring type, is used to support static and instance constructor multi-methods. In the static invocation mode, only methods defined within one class can override for multi-dispatch. Hence, we do not wish to merge static multi-methods from different classes. Instance constructor methods are invoked in the non-virtual modes and operate in the same way. Constructors are not inherited, hence we separate constructors by class. Java only allows methods to be implemented by classes, hence this type field will alway be a classblock in Java. For methods invoked in the virtual and super modes, inherited multi-methods may apply. Hence, separating implementations by class is not useful. So, we merge all overriding methods from any class together by specifying the behaviour's declaring type as `NULL`.

The second value, the selector, separates multi-methods by method name.

The third value that discriminates multi-methods is a terse signature. It is a string of characters that encode the "kind" of value each argument is expected to contain. We have one "kind" for each primitive type, and one representing all object types. In addition to dividing method up by arity, this concise, yet descriptive, signature allows us to efficiently handle primitive arguments as well. Recall that the JVM does not dispatch on primitive arguments, because they have no type hierarchy — each primitive type is isolated and equal. Therefore, there is no need to consider arguments of primitive type when dispatching; they can be skipped over. The terse signature allows us to recognize these non-dispatchable arguments and avoid walking the detailed signature (part of the NAME&TYPE) to determine the argument types.[3] Second, some primitive values — namely `long` and `double` occupy two argument slots at invocation. The signature also encodes this fact as well.

In Figure 5.1, we see the various behaviours generated for a simple program. We recognize two behaviours for constructors, one for each of the `Super` and `Sub` classes. These will be used during non-virtual mode invocations. Another behaviour contains two implementations of `Super.smethod(...)` — the third is not multi-dispatched because it accepts no object arguments. Two more behaviours are generated for the two groups of `*.method(...)` methods. One contains implementations that accept only a single object argument (which includes array arguments) and the other accepts an int argument and a single object. The single int argument is not multi-dispatched since the argument is of primitive type. Their signatures provide the discriminating feature.

Merging Multi-Methods

As an aside, we explore the ramifications of merging all virtual multi-methods with the same selector and signature. An issue arises regarding compatibility of these implementations. We could have two disjoint type hierarchies that override on the `*.add(Object)` multi-method. For example, a type hierarchy starting at `List`, and continuing with `SortedList`, `Vector`, etc. might use `add` to mean insert a new element. Another type hierarchy, starting at `Magnitude`,

---

[3] Primitive values are not self-describing.

```
// simple argument type hierarchy
class A { ... }
class B extends A { ... }
class Super implements
      VirtualMultiDispatchable,
      StaticMultiDispatchable,
      SpecialMultiDispatchable {
  // constructors
  Super() { ... } // !MMD
  Super(A a) { ... }
  Super(B b) { ... }
  Super(int i) { ... } // !MMD
  // static methods
  static smethod(A a) { ... }
  private static smethod(A[] a) { ... }
  static smethod(int i) { ... } // !MMD
  // instance methods
  method(int i) { ... } // !MMD
  method(int i, A a) { ... }
  method(int i, B b) { ... }
  method(A a) { ... }
  method(A[] a) { ... }
}
class Sub extends Super implements
      StaticMultiDispatchable,
      SpecialMultiDispatchable {
  // constructors
  Sub() { ... } // !MMD
  Sub(A a) { ... }
  Sub(B b) { ... }
  // static method
  static smethod(B b) { ... }
  // instance methods
  method(A a) { ... }
  method(B b) { ... }
  private method(Super s) { ... }
  method(int i, B b) { ... }
}
```

(a) Sample Program

```
Type:             Super
Selector:         <init>
Signature:        -O
Implementations:  Super(A)
                  Super(B)
-----------------------------------------------
Type:             Sub
Selector:         <init>
Signature:        -O
Implementations:  Sub(A)
                  Sub(B)
-----------------------------------------------
Type:             Super
Selector:         smethod
Signature:        -O
Implementations:  smethod(A)
                  smethod(A[])
-----------------------------------------------
Type:             Sub
Selector:         smethod
Signature:        -O
Implementations:  smethod(B)
-----------------------------------------------
Type:             null
Selector:         method
Signature:        OIO
Implementations:  Super.method(int, A)
                  Super.method(int, B)
                    Sub.method(int, B)
-----------------------------------------------
Type:             null
Selector:         method
Signature:        OO
Implementations:  Super.method(A)
                  Super.method(A[])
                    Sub.method(A)
                    Sub.method(B)
                    Sub.method(Super)
-----------------------------------------------
Not in any behavour:
                Super()
                Super(int)
                Super.smethod(int)
                Super.method(int)
                Sub()
```

(b) Behaviours Generated

Figure 5.1: Behaviour Management

and continuing on with `Number`, `FixedPointNumber`, and `BigNum` might use `add` to represent an arithmetic operation. Because the original methods in each hierarchy, `List.add(Object)` and `Magnitude.add(Object)`, are unrelated, merging the two sets of methods may appear problematic. The issue is one of *aliasing* — the same selector, `add` is used to mean two completely different operations.

Yet, our choice of `NULL` as the declaring type for virtual methods means that both method sets will be merged into one behaviour. There is no conflict, because the receiver hierarchies are disjoint. The only way unrelated classes (i.e. neither is a subtype of the other) can implement the same behaviour is through both implementing an interface method.[4] But, interfaces cannot appear as receivers — all methods are defined in classes only. Therefore the conflict is moot.

---

[4] Interfaces are a way of expressing design decisions, not implementation.

There is one place where merging all virtual multi-methods with the same selector and signature can result in a performance penalty. That is in a tool, similar to `MDLint` which intends to statically check that overriding (in the multi-method sense) implementations are compatible. `MDLint` must ensure that for every overriding method, its visibility is equal or greater than the method it overrides (to flag potential `IllegalAccessErrors` at compile time), that all conflict methods are defined (to flag potential `AmbiguousMethodErrors` at compile time), that return types are compatible (to flag `IllegalReturnTypeErrors` at compile time).

However, for unrelated multi-methods, these static checks do not apply. If `MDLint` merges the unrelated sets of methods, then it must check every pair for an overriding relationship before applying the static check. A more efficient implementation separates the two unrelated sets of multi-methods into separate behaviours, by using the actual receiver of the base multi-method (which all other multi-methods override) as the base type. Then all methods in the behaviour are known to have the overriding relationship, as needed for the static checks.

All behaviour management routines are protected by a lock, `BEHAVIOUR_LOCK`. These routines include `RegisterImpl()` which adds a new implementation to a behaviour, and `ExtendBehaviour()` which widens the `bits` array to hold a new column for the new type. In addition, each behaviour contains a lock which is acquired whenever that specific behaviour is being updated. Therefore, behaviours are not accessed except in a consistent state.

In addition to these "key" fields, which uniquely identify each behaviour, other fields which support the actual dispatch operations comprise a behaviour. These fields, forming the implementation registration system, include:

`dispatchedSlots` a 32-bit bitfield, one bit for each argument slot (including the receiver) indicating whether that argument is dispatchable.[5][6] Object arguments are represented by `1`, primitives by `0`, beginning with the the receiver in the least significant bit position. Primitives which require two slots, `double` and `long`, occupy two consecutive bit positions.

`impls[]` an array of methodblock pointers to the implementations for this behaviour,

`overrides[]` an one-dimensional array of bitfields, one for each implementation. If bit codem of `overrides[n]` is cleared, then implementation `n` overrides implementation `m`.

`bits[][]` a two-dimensional array of bitfields, one row for each dispatchable argument and one column for each assigned type-number (described in the next section). Each bitfield contains one bit for each implementation in the behaviour. If `bits[r][t]` has bit `i` set, then implementation `i` accepts arguments with type-number `t` at dispatchable argument number `r`.

---

[5] Our limitation of 32-arguments is not unduly constricting, the largest slot count we have encountered is 21 for `MotifGraphicsUtils.layoutMenuItem(...)` of which only 15 are objects. Its signature is `OOOOOOOOIIIIOOOOOOOII`.

[6] Recall that for static and constructor methods the receiver class is statically known and does not need to be dispatched.

`unresolved` a bitfield, with one bit for each implementation, which indicates whether all of that implementation's arguments are resolved.

`toresolve` a bitfield, with one bit for each implementation, which indicates which method must be resolved in order to complete a dispatch. The use of this field is discussed in Section 5.2.2.

We illustrate the structure of a behaviour by giving the details for the virtual behaviour containing `Super.method(int, A)`. For notational convenience, we will name that behaviour `*.method(int,*)`, where `*` indicates an non-fixed object argument. We further assume for the purposes of illustration, that classes `Super`, `Sub`, and `A` have been numbered (0), (1), and (2). Last, we assume that class `B` has not yet been loaded. The structure is given in Figure 5.2, where all bitfields are expressed in binary with most-significant bit on the left and truncated to three bits for clarity (except `argsresolved`)[7].

Type: `null`

Selector: `method`

Signature: `OIO`

Dispatched Slots: `101`

Implementations:	Overrides:	*Arg Type-Nums*	*Args Resolved*
`Super.method(int, A)`	`111`	*[0, -4, 2]*	*000*
`Sub.method(int, B)`	`111`	*[1, -4, U]*	*100*
`Super.method(int, B)`	`111`	*[0, -4 U]*	*100*

Bits:

	TypeNumbers			
	Super	*Sub*	*A*	*B*
	0	*1*	*2*	*U (unresolved)*
row 0	`101`	`111`	`000`	
row 1	`110`	`110`	`111`	

Unresolved: `110`

To Resolve: `000`

Figure 5.2: Behaviour `*.method(int,*)`

The shaded portions show information which is not contained in the behaviour itself. The argument type numbers and resolved flags for the implementations are contained in the methodblocks. The class type numbers are stored in the classblocks, and in a global array `classesByTypeNum` for quick access. We have also deliberately placed the methods in a different order for use in later illustrations.

---

[7] This is an artifact of the implementation.

The `dispatchedSlots` field indicates that at dispatch time, we will have two arguments to examine: the receiver (rightmost bit) and another two slots down (leftmost bit). Our `overrides` bits indicate that none of the implementations override another: without knowledge of `B`'s place in the type hierarchy, this is correct.

Examining the dispatch `bits` next, we recognize that if the receiver (row 0) is a `Super`, then all three implementations apply. If the receiver is a `Sub` then only the second implementation applies, and no implementations apply if the receiver is of type `A`. With regard to the other dispatchable argument (row 1), we see that second and third implementations always apply — this is because we don't have accurate type information about this argument for those implementations. We also recognize that the first implementation only applies when the second dispatchable argument is an `A`.

The second and third implementations are not fully resolved: the `unresolved` field stores this fact. The `toresolve` bitfield is currently empty.

The last detail about behaviours is their size. We constructed four sizes (originally eight) to contain differing numbers of implementations. Behaviours can contain 32, 64, 128, and 256 implementations. Previously we supported 8 and 16 implementation behaviours, but memory alignment issues eliminated most of the benefits of these smaller bitfield sizes. We also originally supported 512 and 1024 implementation behaviours, but benchmarks showed that the 256 implementation behaviour structure was never used. Behaviours are automatically resized when the over-capacity implementation is registered.

Type-Numbering

The type-numbering algorithm is designed to assign a *unique* integer to each dispatchable argument type. That integer will be used to index into a table of bitfields to determine which method implementations apply.

A set of special type-numbers are initialized at JVM startup. They represent `null`, *unresolved* and primitive types. Because we will not need to index into a behaviour with them, we use negative type-numbers for these. As we will see, `null` arguments give us no information about applicable methods, so they can be assigned a negative type-number, `TYPE_NULL`. We wish to have some unique type-number available for types we have a name for, but have not yet loaded. This is another negative type-number, `TYPE_UNNUMBERED`. Representing primitive types with negative type-numbers is justified because primitive types do not participate in multi-dispatch — in a sense, the behaviour management portion has already dispatched on them. As an aside, one could assign a single `TYPE_PRIMITIVE` to be shared by all of the primitive types, but we choose to differentiate them for some future use.

Every other type-number is zero or greater, and is used to designate a reference type — a class or interface.

We define a new field `type_num` as part of the classblock to contain the type-number. At class-loading time, each class and interface is assigned a type-number. By default, that number is `TYPE_UNNUMBERED`, a unique value less than zero, indicating that the class or interface has not been assigned (or inherited) its own type-number. Two conditions can result in a different type-number being assigned. First, if the class or interface being loaded is a subtype of a numbered type, then it receives its super-type's type-number instead. For the purposes of type-numbering, a class is a subtype of both its direct superclass and any interfaces it directly implements. Second, if the class being loaded has multiple super-types, and they comprise more than one type number (other than `TYPE_UNNUMBERED`), then the loaded class is assigned a new, unique type-number.

Array types must be correctly type-numbered as well; but they are not loaded from classes. Instead the JVM creates "fake array classblocks" that can be looked up by name. Their type hierarchy matches that shown in Figure 3.8. In particular, `Object[]` inherits its type-number from `Serializable` and `Cloneable`. If neither interface has a unique type-number, then `Object[]` will be given `TYPE_UNNUMBERED`. If only one of two interfaces have an assigned type-number, then `Object[]` will receive that type-number. If both have been assigned or inherited type-numbers, then `Object[]` will be given a new type-number.

Implementing this array type numbering was a complex endeavour. As we noted previously, multi-dispatch forces us to support parametric polymorphism using dynamic dispatch on the parameter type.

Our original implementation design exploited the parallel structure of the type hierarchy in each array dimension by endowing each type with an array of type-numbers, one for each array dimension. Simple investigation showed that array types tended to have low dimension — for instance, the Java 1.3 class hierarchy never exceeds two-dimensional arrays — for example `Byte [][]` for image rasters.

Further, the hierarchy of types used to parameterize arrays tended to be small — it rarely showed the same depth and diversity that the basic class/interface hierarchy showed. In retrospect, this is clear from the container problem: arrays were typically constructed to hold objects of a general type, acting as a container for many other more specific types. For example, `Event[] eventlist` rather than `MouseMovementEvent[] eventlist`. In the rare instance where an array was parameterized by a very specific type, the more general types never appeared as array parameters. For example, `JButton[] buttons` appeared, but not `JFileChooser[]`, `JButtonBar[]`, etc.

As a result, we elected to force every array superclass to also be instantiated in order to avoid gaps in the type hierarchy when type numbering. This has worked surprisingly well, as we shall see in Chapter 6.

When a multi-dispatch method is recognized, all of its argument types (including the receiver) are verified to have a unique type number. There are three possible cases for each argument:

1. the class, interface, or array for the argument is not yet loaded: the classblock is assigned `TYPE_UNNUMBERED` representing an unresolved type,
2. the class, interface, or array already has been assigned a unique type number: nothing needs to be done,
3. the class, interface, or array has inherited a shared type-number from a superclass: in this case, the classblock is assigned a new, fresh type-number and that type-number is propagated to all of the subtypes of that classblock.

In the last case, type-numbers are propagated recursively along the subclass chain, until either no more subclasses remain, or a subclass with its own unique type-number is encountered.

As an example, consider the following sequence of type-numbering events, illustrated in Figure 5.3.

1. The JVM is initialized, `Object` exists, with `TYPE_UNNUMBERED` (shown as (U)).
2. Class `C` is loaded, which forces its superclass, `A` and superinterface `I` to be loaded. All inherit type-number (U).
3. Class `A` is forced to be numbered by appearing in a multi-method `A.mmd(I)`. It obtains type-number (0), which propagates to `C`.
4. Interface `I` is forced to be numbered by also appearing in multi-method `A.mmd(I)`. It obtains type-number (1). This forces `C` to obtain the unique type-number (2) because it now inherits two different type-numbers.
5. Class `D` is loaded, it inherits its type-number from the super-interface `I`, since `Object` has type-number (U).
6. Class `E` is loaded, and it inherits type-number (2) from its superclass `C`.
7. Class `D` appears as an argument to `D.mmd(I)`. This forces class `D` to be assigned its own unique type-number (3).
8. Class `B` is loaded, and it inherits type-number (0) from its superclass `A`.

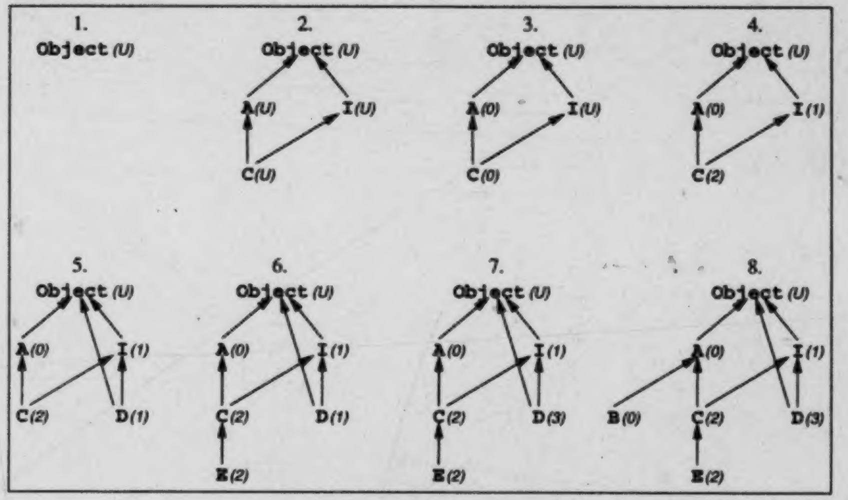


Figure 5.3: Type-Numbering Operations

All type-numbering operations occur while the JVM `BINCLASS_LOADER` lock is held, and so are serialized to ensure consistency. The type-numbering scheme ensures that every type is assigned a unique type-number at most once — a type never changes its unique type-number. Any loaded type that appears as a receiver or argument to a multi-method will have a unique type-number. Only types that must be distinguished will have unique-type numbers. In particular, subtypes which only respond to the same multi-methods as their super-type will inherit their super-type-number. This holds true for a subtype of two differently type-numbered types — it must respond to multi-methods from each super-type.

To further ensure consistency, type-numbering operations occur during the linking phase of class loading [32]. Since a class cannot be accessed until it is fully linked into the JVM runtime, there is no possibility of a newly allocated type-number being used erroneously. Also, the JVM ensures that classes are only added to the internally-known class list once, so it is impossible for a class to be assigned two type numbers. Array classes are registered and type-numbered within `createFakeArrayClass`.

Every behaviour contains an array of dispatch bitfields, indexed by type-number. Hence, when a new type-number is allocated, each behaviour must be visited and extended to support that new type-number. As we shall see below, the bitfields for a new type-number are initialized to the "bitwise-or" of its super-types, or to the unresolved method bits if no super-type has been numbered.

The methodblock of each multi-method contains an array of type-numbers, one for the receiver and one for each argument. If a type-number cannot be found, because an argument type has not yet been loaded, a flag in the methodblock is set indicating that fact, and the missing argument type is assigned `TYPE_UNNUMBERED`. These multi-methods which lack a complete set of argument type-numbers are called *unresolved*, since their argument types belong to classes that have not yet been linked into the runtime.[8]

Figure 5.4 shows the changes to Figure 5.2 `*.method(int,*)` method when class `B` is loaded and type-numbered. Note that none of the methods have had their arguments resolved, that will be done if those incompletely resolved methods ever apply in a multi-method dispatch.

Type: `null`

Selector: `method`

Signature: `OIO`

Dispatched Slots: `101`

Implementations:	Overrides:	*Arg Type–Nums*	*Args Resolved*
`Super.method(int, A)`	`111`	*[0, -4, 2]*	*000*
`Sub.method(int, B)`	`111`	*[1, -4, U]*	*100*
`Super.method(int, B)`	`111`	*[0, -4 U]*	*100*

Bits:

	TypeNumbers			
	Super	***Sub***	***A***	***B***
	0	*1*	*2*	*3*
row 0	`101`	`111`	`000`	`000`
row 1	`110`	`110`	`111`	`111`

Unresolved: `110`

To Resolve: `000`

Figure 5.4: Behaviour `*.method(int,*)` After Numbering `B`

Registering Implementations

The SRP dispatch algorithm needs to have the property that an implementation that overrides others must be at a higher (more significant) bit position than those it overrides. To this end, a `moveImpl()` routine permits an implementation to migrate up or down in the list of implementations and modifies all of the related bitfields. The `bits` bitfields for each dispatchable argument (row) and type-number (column) have the old bit-value deleted from the old position and inserted at the new position. The `unresolved`, `toresolve` bitfields, and each `overrides` bitfield is similarly updated.

Adding an implementation calls the `setDispatchBits()` routine, responsible for updating the `overrides` and `bits` fields in the behaviour. When a new implementation is added to a behaviour, it is compared for specificity against each of the implementations already registered with the behaviour. It is inserted in the next higher position after the last implementation it overrides. There is a potentially confusing issue here — what to do with arguments which are currently unresolved? For the purposes of deciding the method order, unresolved arguments are presumed to override everything. This is called *weak overriding*. We apply the term *strict* overriding where unresolved arguments override nothing.[9] Hence, implementations will always be placed at the most significant possible bit position.

---

[8] Recall that the process of linking a symbolic reference to a class or method is called resolution.

Once the implementation's position `i` is determined, each bitfield in the behaviour `bits` field is visited to decide whether its bit position `i` should be set. `Bit[r][t]` is enabled if argument slot [r] of the method M_i contains type number `TYPE_UNNUMBERED`, or argument slot [r] of M_i is a subtype of the type numbered `t`. The `overrides` bits are also examined. For each other method M_j, if M_i strictly overrides M_j, then bit `j` of `overrides[i]` is cleared. If M_j strictly overrides M_i then bit `i` of `overrides[j]` is cleared. The cleared bits in `overrides[i]` indicate which implementations method M_i is maximally specific over.

Resolving methods also forces the `setDispatchBits()` procedure to be called. The effect of resolving both unresolved methods from Figure 5.4 is given in Figure 5.5. Note that the order of methods has changed to recognize that `Sub.method(B)` $\preceq$ `Super.method(B)`. The overrides bits are updated to recognize the overriding relationships among the methods as well.

Type: `null`

Selector: `method`

Signature: `OIO`

Dispatched Slots: `101`

Implementations:	Overrides:	*Arg Type–Nums*	*Args Resolved*
`Super.method(int, A)`	111	*[0, -4, 2]*	*000*
`Super.method(int, B)`	110	*[0, -4, 3]*	*000*
`Sub.method(int, B)`	100	*[1, -4, 3]*	*000*

Bits:	*TypeNumbers*			
	Super	***Sub***	***A***	***B***
	0	*1*	*2*	*3*
row 0	011	111	000	000
row 1	000	000	001	111

Unresolved: `000`

To Resolve: `000`

Figure 5.5: Behaviour `*.method(int,*)` After Resolving All Methods

---

[9] A method can only *strictly override* another iff both methods are completely resolved.

SRP Dispatcher

Finally, we can examine the dispatch algorithm itself. The basic SRP algorithm is straightforward:

Step 1. Walk the `dispatchedSlots` bits, obtaining the type-number `t[r]` for each dispatchable argument `r`.

Step 2. Begin with a bitfield `mbits` with all implementations in the behaviour enabled.

Step 3. For each dispatchable argument `r` of type `t[r]`, mask off the implementations that are not applicable, by "bitwise-and"-ing `mbits & bits[r][t[r]]` into `mbits`.

Step 4. Determine the index of the most specific applicable implementation by finding `i`, the most-significant bit set in `mbits`.

Step 5. Return the implementation `impls[i]` as the multi-dispatched methodblock.

For example, consider multi-dispatching the invocation of `method` via `(new Super()).method(1,new A())` from Figure 5.5. We build an array of type-numbers for the two dispatchable arguments, `[0, 2]`. SRP starts with a bitfield containing all implementations `111`. Next, SRP "bitwise-and"s the applicable methods for the receiver, from row 0, type-number 0, giving `111 & 011 = 011`. Next, SRP "bitwise-and"s the applicable methods for the other dispatchable argument, from row 1, type-number 2, giving `011 & 001 = 001`. The most-significant bit set is bit 0, indicating that the first implementation `Super.method(int,A)` is desired.

This basic algorithm does not meet our needs. First, it requires a type-number for type `null`, which (as we shall see) is unnecessary and inefficient. Second, SRP must have complete type information about all of the arguments involved in multi-dispatch. This is reasonable for a statically-linked language such as C++, but not for Java.

`null` arguments

The `null` type has a special property: all object types are a super-type of it. Formally, $null \preceq T$ for all non-primitive types T, see Figure 3.8 for an illustration. Therefore, a `null` value is acceptable anywhere an object is. Hence, every method will accept a `null` for any dispatchable argument. But, this means that an argument of type `null` never reduces the set of applicable methods. Hence, it is pointless to maintain a type-number and column in each `behaviour->bits` structure for it. Therefore, we test for `TYPE_NULL` in Step 3, and do not mask off any bits if it is seen. Step 3 becomes

Step 3. For each dispatchable argument `r` of type `t[r]`,

Step 3a. If `t[r]` is `TYPE_NULL` then do nothing, else

Step 3b. mask off the implementations that are not applicable, by "bitwise-and"ing `mbits & bits[r][t[r]]` into `mbits`.

By not allocating a real type-number and `bits` column to type `null`, our implementation has retained the ability to implement dispatch table compression, for example, selector coloring [2].[10] If `null` occupied a column in every behaviour `bits` array, then that column would never be empty, and `bits` rows from the same (or other) bitfields could never be merged. This optimization saves space, time, and preserves compression opportunities.

Unnumbered arguments

Another way in which Step 3 needs revision is to support arguments that have not been numbered. This can happen if multi-method implementations are registered before all the argument classes are loaded, and those argument classes, when loaded are not multi-dispatchable and hence not forced to be numbered. We see this in Figure 5.2, with class `B`. Step 3a is amended to:

Step 3a. If `t[r]` is `TYPE_NULL` or `TYPE_UNNUMBERED` then do nothing, else

We now focus on the second requirement: SRP must have complete type information. It conflicts directly with the need to incrementally load classes during the execution of the program. As a result, the basic SRP algorithm needs to be modified to permit two new features: first, recognition of ambiguous dispatches at dispatch time rather than at table construction time, and second, dispatch with imprecise type information that loads only those types needed to resolve the imprecision.

Ambiguous Dispatches

The SRP algorithm presumes that at dispatch time, a most specific implementation exists for all possible argument combinations. It does this by verifying that any needed conflict methods are defined. However, this check must happen as methods are inserted into the behaviour. Thus an `AmbiguousMethodError` would be reported at class load time. As discussed previously, we want to be lazy about these checks, and report ambiguous dispatches only when invoked. This is easily done by ensuring that the final bitfield does contain exactly one bit set.

Figure 5.6 shows an ambiguous dispatch for the `Super.smethod(*)` multi-method. With a `null` argument, the dispatch is ambiguous: `A` is not a subtype of `A[]` and vice-versa. A useful feature of our implementation is that `AmbiguousMethodErrors` contain a field `Method[] methods` which contains each of the methods which applied to the ambiguous dispatch;[11]

---

[10] We have not implemented these compression techniques

[11] Our example shows the tricky nature of type `null` in particular there is no way in Java to express the conflict method needed.

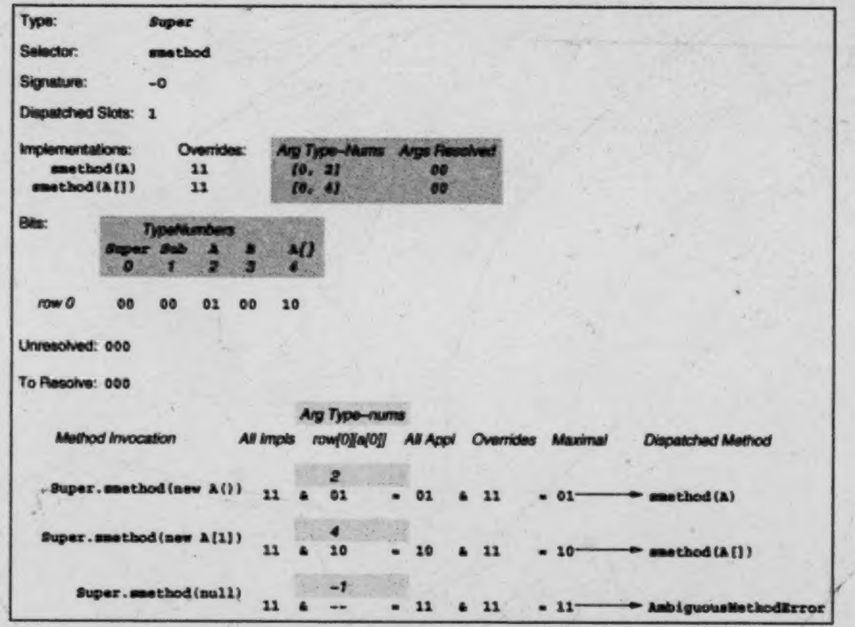

Type: Super
Selector: smethod
Signature: -0
Dispatched Slots: 1
Implementations:
smethod(A)
smethod(A[])
Overrides:
11
11
Arg Type-Nums
[0, 2]
[0, 4]
Args Resolved
00
00
Bits:
TypeNumbers
Super Sub A B A[]
0 1 2 3 4
row 0 00 00 01 00 10
Unresolved: 000
To Resolve: 000
Arg Type-nums
Method Invocation
All impls
row[0][a[0]]
All Appl
Overrides
Maximal
Dispatched Method
Super.smethod(new A())
2
11 & 01 = 01 & 11 = 01
smethod(A)
Super.smethod(new A[1])
4
11 & 10 = 10 & 11 = 10
smethod(A[])
Super.smethod(null)
-1
11 & -- = 11 & 11 = 11
AmbiguousMethodError

Figure 5.6: Ambiguous Dispatch in `Super.smethod(*)`

A more typical example would be the "multi-dispatch diamond" found in the program in Figure 4.8, where `new Sub().method(new Sub())` is ambiguous.

Unresolved Methods

With lazy class-loading, until class `B` is loaded, the JVM, inFigure 5.2, only knows that `A` $\not\preceq$ `B`. This is because if `A` was a subclass of `B`, then `B` would have already been loaded — super-classes are always loaded before subclasses. There are two possibilities as seen in Figure 5.7: `B` and `A` are unrelated, or `B` is a subclass of `A`. Without complete type information, the JVM does not know which conflict methods are required for `*.method(int,*)`. Hence, ambiguous dispatch checks would require loading `B` (and many other) classes when building dispatch tables. We do not want that overhead.

Even if `B` is loaded and type-numbered, previously registered implementations might not have the new type-numbering information. Figure 5.4 shows this: class `B` has been loaded and type numbered, but previously registered implementations `Super.method(B)` and `Sub.method(B)` do not have updated argument type-numbers. They *apparently apply* to any `(new Super()).method(int, null)` invocation. These methods need to updated with the revised type-number information, and their dispatch bits recomputed. But rather than searching out every unresolved method and updating it at every class load operation, we want to perform this updating lazily as well.

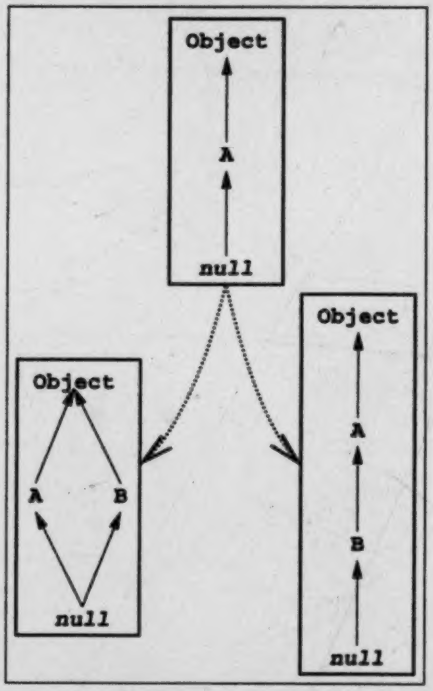


Figure 5.7: Unresolved Types

Our dispatch implementation handles both issues. It forces the loading of any classes needed to resolve (apparently applicable) methods, and revises those methods' dispatch information. Our SRP algorithm has an extended Step 4 to provide this support.

Step 4a. Let `i` be the most-significant bit set in `mbits`. Examine `mbits`, if only `i` is set then the most specific applicable method has been found — go to Step 5 just like the original SRP algorithm.

Step 4b. More than one bit in `mbits` is set, then other methods apply. By the ordering of the implementations, `i` must correspond to a maximally specific method. Remove any methods it overrides by computing `mbits & overrides[i]`. See if any bits remain set in `mbits`, if not then the most specific applicable method has been found — go to Step 5. Otherwise, any remaining bits correspond to other methods that are maximally specific, but not overridden by implementation `i`.

Step 4c. Determine if any remaining implementations remain because they are unresolved, by "bitwise-and"ing `mbits & unresolved`. If the result has no bits set, then we have an ambiguous dispatch because we have two (or more) maximally specific implementations. Throw an `AmbiguousMethodError`.

Step 4d. Some unresolved methods apply; resolving these may discover a most-specific applicable method. Note the implementations that need resolution by copying `mbits` into `toresolve`. Resolve those methods, and retry the dispatch.

We show the operation of the dispatcher and the contents of the behaviour structure for `*.method(int,*)` as we progress from Figure 5.2 through a sequence of dispatches.

			Arg Type-nums										
Method Invocation	*All Impls*		*row[0][a[0]]*		*row[1][a[1]]*		*All Appl*		*Overrides*		*Maximal*		*Dispatched Method*
`(new Super()).method(1, new A())`			*0*		*2*								
	111	&	101	&	111	=	101	&	111	=	101	→	`resolve Super.method(int,B)`
`(new Super()).method(1,new B())`			*0*		*U*								
	111	&	101	&	---	=	101	&	111	=	101	→	`resolve Super.method(int,B)`
`(new Super()).method(1,null)`			*0*		*N*								
	111	&	101	&	---	=	101	&	111	=	101	→	`resolve Super.method(int,B)`
`(new Sub()).method(1,new A())`			*1*		*2*								
	111	&	111	&	111	=	111	&	111	=	111	→	`resolve Sub.method(int,B)` `and Super.method(int,B)`
`(new Sub()).method(1,new B())`			*1*		*U*								
	111	&	111	&	---	=	111	&	111	=	111	→	`resolve Sub.method(int,B)` `and Super.method(int,B)`
`(new Sub()).method(1,null)`			*1*		*N*								
	111	&	111	&	---	=	111	&	111	=	111	→	`resolve Sub.method(int,B)` `and Super.method(int,B)`

Figure 5.8: Dispatches for Figure 5.2

We attempt to dispatch `(new Super()).method(1,new A())` (shown in Figure 5.8), this forces the resolution of `Super.method(int,B)`. The resulting behaviour structure is given in Figure 5.9.

The potential dispatches are given in Figure 5.10.

If we dispatch `(new Sub()).method(1,new A())` we must resolve the last unresolved method, and obtain the behaviour structure given in Figure 5.5. The potential dispatches from that state are shown in Figure 5.11.

Bitfield Operations

We use a reasonably optimized set of operations to test, set, clear, copy, insert, delete and extend bitfields. As noted previously, we work with 32, 64, 128, and 256 bit wide fields. One time-intensive operation, the 32-bit *find-first-set*, has been coded in I686 assembler for Intel Pentium-III CPUs. It is lightly edited version of the system library version found in `glibc-2.2.5` [19]. In particular, it requires exactly two machine instructions, but an unknown number of clock cycles. Non-PIII systems can use a simple unrolled binary search implementation.

We conclude our implementation section with the pseudo-code for the dispatcher in Figure 5.13. The actual C code is given in Section E.2.

Type: `null`

Selector: `method`

Signature: `010`

Dispatched Slots: `101`

Implementations:	Overrides:	*Arg Type–Nums*	*Args Resolved*
`Super.method(int, A)`	`111`	*[0, -4, 2]*	*000*
`Sub.method(int, B)`	`111`	*[1, -4, U]*	*100*
`Super.method(int, B)`	`110`	*[0, -4, 3]*	*000*

Bits:

	TypeNumbers			
	Super	*Sub*	*A*	*B*
	0	*1*	*2*	*3*
row 0	`101`	`111`	`000`	`000`
row 1	`010`	`010`	`011`	`110`

Unresolved: `010`

To Resolve: `000`

Figure 5.9: After Dispatching `(new Super()).method(1,new A())`

Method Invocation	*All Impls*		*Arg Type-nums* *row[0][a[0]]*		*row[1][a[1]]*		*All Appl*		*Overrides*		*Maximal*		*Dispatched Method*
`(new Super()).method(1, new A())`			*0*		*2*								
	111	&	101	&	011	=	001	&	111	=	001	→	`Super.method(int,A)`
`(new Super()).method(1,new B())`			*0*		*3*								
	111	&	101	&	110	=	100	&	110	=	100	→	`Super.method(int,B)`
`(new Super()).method(1,null)`			*0*		*N*								
	111	&	101	&	---	=	101	&	110	=	100	→	`Super.method(int,B)`
`(new Sub()).method(1,new A())`			*1*		*2*								
	111	&	111	&	011	=	011	&	111	=	011	→	`resolve Sub.method(int,B)`
`(new Sub()).method(1,new B())`			*1*		*3*								
	111	&	111	&	110	=	110	&	110	=	110	→	`resolve Sub.method(int,B)`
`(new Sub()).method(1,null)`			*1*		*N*								
	111	&	111	&	---	=	111	&	110	=	110	→	`resolve Sub.method(int,B)`

Figure 5.10: Dispatches for Figure 5.9

Multi-Threading Support

Unlike the MSA algorithm, the SRP data structures are globally shared, containing information about many classes and methods. Therefore, different threads may need to gain access to the same `behaviour` structure simultaneously. This leads to concurrency issues, primarily surrounding the creation of `behaviour` structures, the updating of `behaviour` structures with newly loaded method implementations, and the forced resolution of implementations upon

Method Invocation	All Impls		Arg Type-nums row[0][a[0]]		 row[1][a[1]]		All Appl		Overrides		Maximal		Dispatched Method
`(new Super()).method(1, new A())`			0		2								
	111	&	011	&	001	=	001	&	111	=	001	→	`Super.method(int, A)`
`(new Super()).method(1,new B())`			0		3								
	111	&	011	&	111	=	011	&	110	=	010	→	`Super.method(int, B)`
`(new Super()).method(1,null)`			0		-1								
	111	&	011	&	---	=	011	&	110	=	010	→	`Super.method(int, B)`
`(new Sub()).method(1,new A())`			1		2								
	111	&	111	&	001	=	001	&	111	=	001	→	`Super.method(int, A)`
`(new Sub()).method(1,new B())`			1		3								
	111	&	111	&	111	=	111	&	100	=	100	→	`Sub.method(int, B)`
`(new Sub()).method(1,null)`			1		-1								
	111	&	111	&	---	=	111	&	100	=	100	→	`Sub.method(int, B)`

Figure 5.11: Dispatches for Figure 5.5

multi-dispatch. Roughly, these three problem areas correspond to managing the hashtable of `behaviours`, managing the implementations in a `behaviour`, and protecting the `toresolve` bits in a `behaviour`.

The first data structure which needs concurrent access protection is the `behaviour` hashtable. The hashtable is only updated at class-load time, so we accept the relatively heavy-weight operation of entering a monitor, the `BEHAVIOURS_LOCK` whenever a new behaviour is being created. To ameliorate the penalty, we use the double-checked locking idiom [39] in `LookupBehaviour`. In most cases, the behaviour already exists, and the lock does not need to be acquired.

The second data structure which needs concurrent access control is the `behaviour`. To provide this protection, every `behaviour` contains a monitor.[12]. For simplicity, we implemented a `MMD_SIMPLE_LOCKING` system, where every `behaviour` is guarded for updates and dispatches by entering the monitor. If additional resolution is needed to multi-dispatch an ambiguous invocation, the thread remains in the monitor during that process. But, this is more restrictive than it needs to be. In particular, a multi-dispatch does not alter the

---

[12] We do provide a conditional compilation flag, `MMD_NO_LOCKING`, which disables `behaviour` locking, but we do not recommend its use except for single-threaded applications

```
inline int ffs32(int x) {
  /* assumes i686 */
  /* extracted from glibc-2.2.5, modified scan in reverse beginning with 0 */
  int cnt;
  int tmp;
  asm ("bsrl %2,%0"          /* Count low bits in X and store in %1.  */
       "cmovel %1,%0"        /* If number was zero, use -1 as result.  */
       : "=&r" (cnt), "=r" (tmp) :  "rm" (x), "1" (-1));
  return cnt;
}
```

Figure 5.12: Find-First-Set Implementation

```
proc Method DISPATCH(JavaStack stack, Behaviour b) {
    bitfield mbits, sbits;
    int row, slot, r;
    typenums[] atnums;
    int i;
  LOOP:
    // Step 1 - get argument type-nums
    for (sbits = b.dispatchedSlots, slot=0, row=0;
         slot < b.nDispatchableSlots;
         sbits >>= 1, slot++) {
        if (sbits & 1)
            atnums[row++] = PEEKTYPE(stack, slot);
    }
    // Step 2 - initial bitfield
    mbits = -1;
    // Step 3 - mask off inapplicable methods
    for (r=0; r<row; r++)
        switch atnums[r] {
            case TYPE_NULL:
            case TYPE_UNNUMBERED: continue;
            default:  mbits &= bits[r][atnums[r]];
        }
    // Step 4a - examine resulting bitfield
    i = FFS(mbits);
    mbits &= ~(1 << i);
    if (mbits == 0) return impls[i];
    // Step 4b - remove overrides and reexamine
    mbits &= overrides[i];
    // Step 5 - only one method
    if (mbits == 0) return impls[i];
    // Step 4c - check for unresolved methods
    toresolve = mbits & unresolved;
    // no methods to resolve => but ambiguous
    if (toresolve == 0)  THROW(AmbiguousMethodError);
    // Step 4d - resolve methods, redispatch
    while (toresolve != 0) {
        r = ffs(toresolve);
        RESOLVE(impls[r]);
        toresolve &= ~(1 << r);
    }
    goto LOOP;
}
```

Figure 5.13: Extended SRP Dispatcher

`behaviour` structure, so concurrent reads should be permitted.

Therefore, we have implemented a concurrent-read–exclusive-write mechanism where the `behaviour` contains a volatile boolean flag indicating whether its tables are consistent. If that flag is not set, the multi-dispatching thread continues unimpeded. If the volatile flag is set, then the multi-dispatching thread must enter the `behaviour` monitor. Since the updating thread will be in the monitor until it is finished updating the `behaviour`, a multi-dispatching thread will pause until the `behaviour` structure is consistent and the updating thread has exited the monitor. It is the responsibility of the updating thread to ensure that no multi-dispatches are in-progress when the monitor is entered. At this time, that check is not in place, but appears simple to implement using thread-local storage. This more sophisticated locking technique is controlled by the `MMD_CREW_LOCKING` conditional compilation flag.

The third data structure which needs protection across multi-dispatches is the `toresolve` bitfield. It indicates which implementations must be resolved before a currently ambiguous multi-dispatch can complete. The `MMD_SIMPLE_LOCKING` implementation already ensures concurrency protection by entering the `behaviour` monitor for every multi-dispatch. For `MMD_CREW_LOCKING` we rely on the observation that resolution of an individual implementation can occur only once. Therefore, we expect that contention for the `toresolve` bitfield will be small, occurring early on in program execution. For this reason, we chose not to make `toresolve` a thread-local variable, but placed it as a field of the `behaviour`. But, this means that any dispatch which discovers that methods must be resolved needs to exit the `behaviour` monitor, then enter the `BEHAVIOURS` global monitor and re-enter the `behaviour` monitor before setting the `toresolve` bits, and remain in both monitors until the multi-dispatch completes successfully. The ordering of monitor entry ensures deadlock cannot occur. Our evaluation, as described in Chapter 6, shows that this appears to be a reasonable approach.

With the details of our implementations presented, we now turn our attention to their effectiveness.

Chapter 6

Multi-Dispatch Java - Evaluation

We set out to develop a conservative extension to Java, which would support programmer-targeted multi-dispatch while retaining the syntax, semantics, and performance for existing uni-dispatch code. We evaluate our system based on two criteria: compatibility and performance. The first criterion, compatibility, demonstrates that we have maintained the Java syntax, and the Java semantics for uni-dispatch. Further, we will show that our extensions apply negligible overhead to uni-dispatch applications.

The second criterion, performance, demonstrates that our multi-dispatch implementation out-performs equivalent double-dispatch code in normal Java. We will also examine some of the software engineering advantages that multi-dispatch code has over the equivalent double-dispatch programming.

All experiments were executed on a dedicated Intel-architecture PC equipped with a single 1.0GHz Pentium III processor, a 133MHz front-side bus, and 512 MB of memory. The operating system is Linux 2.4.16 with `glibc` version 2.2.5. The Sun Microsystems' Linux JDK 1.3.0 code was compiled using the GCC compiler version 3.0.2, with optimization flags as supplied by Sun Microsystems' makefiles.[1]. The Sun Microsystems JDK only supports the `green` threading model, which is implemented using pthreads under Linux. We report average and standard deviations for 20 runs of each benchmark.

Our evaluation will examine six different multi-dispatch Java Virtual Machines. They are grouped into two sets of three implementations: we have three different multi-dispatchers, and two different ways of interposing multi-dispatch into the JVM. The six multi-dispatch JVMs are:

MSA-MI an implementation of the MSA algorithm, using the multi-invoker,

SRP-L-MI our tuned SRP algorithm, using the multi-invoker, and applying simple locking,

---

[1] Typical flags are `-O2`

SRP-C-MI our tuned SRP algorithm, using the multi-invoker and applying our concurrent-read–exclusive-write locking,

MSA-INL an implementation of the MSA algorithm, using inlined multi-method flag testing,

SRP-L-INL our tuned SRP algorithm, using inlined multi-method flag testing, and applying simple locking,

SRP-C-INL our tuned SRP algorithm, using inlined multi-method flag testing, and applying our concurrent-read–exclusive-write locking.

These are summarized in Table 6.1.

	Multi-Method Interposition		Algorithm		Locking	
Implementation	Multi-Invoker	Inline Tests	MSA	SRP	Simple	CREW
MSA-MI	*		*			
SRP-L-MI	*			*	*	
SRP-C-MI	*			*		*
MSA-INL		*	*			
SRP-L-INL		*		*	*	
SRP-C-INL		*		*		*

Table 6.1: Multi-Dispatch Implementations

For comparison purposes, we will occasionally introduce performance numbers for other uni-dispatch JVMs. In particular, the ones we will consider are:

UNI the base JDK 1.3.0 JVM compiled without any multi-dispatch enhancements,

CLASSIC the Sun Microsystems JDK 1.3.1 production (binary) JVM running in "classic" mode, with no JIT compiler,

CLIENT the Sun Microsystems JDK 1.3.1 production JVM running in "client" mode with the HotSpot JIT compiler,

SERVER the Sun Microsystems JDK 1.3.1 production JVM running in "server" mode with the HotSpot JIT compiler.

These are summarized in Table 6.2.

Implementation	Mode	JIT Compiler
UNI	classic	none
CLASSIC	classic	none
CLIENT	client	HotSpot
SERVER	server	HotSpot

Table 6.2: Uni-Dispatch Implementations

Our results are organized around a number of comparative tables; we describe the content of those tables in Table 6.3. We include dispatch-level and application-level measurements for uni-dispatch and multi-dispatch, as well as comparisons of the performance of a variety of double-dispatch techniques with multi-dispatch implementations. We also investigate the scaling properties of the various double and multi-dispatch implementations as arity increases.

Table	Level		Note	
Number	Application	Dispatch	Comparison	Note
Table 6.4	*		various JVMs	uni-dispatch absolute times
Table 6.5		*	various JVMs	uni-dispatch absolute times
Table 6.6		*		double dispatch baseline times
Table 6.7		*		multi-dispatch absolute times
Table 6.8		*	double vs multi-dispatch	MSA speed-ups
Table 6.9		*	double vs multi-dispatch	SRP-L speed-ups
Table 6.10		*	double vs multi-dispatch	SRP-C speed-ups
Table 6.11		*	double vs multi-dispatch	scaling properties with arity
Table 6.13	*		double vs multi-dispatch	absolute times

Table 6.3: Summary of Results Tables

6.1 Compatibility

We claim that our system requires no changes in Java syntax, and for uni-dispatch Java we impose no semantic changes nor operational overhead.

Self Compilation

The first guarantee of this comes from the actual construction of the Java Development Kit (JDK) 1.3.0. Part of the process of building the JDK is to compile all of the classes that are included with the JDK. Each multi-dispatch JVM hosts the Java compiler, `javac`, to construct all of the classes for the entire JDK, including the `java.lang`, `java.io`, `java.awt`, `javax.swing`, and `java.util`. In addition, each multi-dispatch JVM executes the Java applications that compute JNI headers, JavaBean information, and the entire javadoc output. Each of our multi-dispatch JVMs successfully complete this entire process, `make release-binaries`, from a source tree that has been cleaned via `make world-clobber`.

Each multi-dispatch JVM has been tested with a number of uni-dispatch Java applications and demonstrations included with the JDK. These include `javap` which displays details of classfiles; `jdb`, the Java debugger; and, `SwingSet2`, the demonstration of the various `Swing` facilities. Each multi-dispatch JVM successfully executed each uni-dispatch application.

With basic compatibility assured, we now consider the performance overhead applied by each multi-dispatch JVM. We provide three experiments — one based on application-level timing, the other two concentrating on uni-dispatch execution times alone.

Javac Timing

The first quantitative experiment requires the runtime to load and execute the `javac` compiler to translate the entire `sun.tools` hierarchy of Java 1.3.0 source files into `.class` files. This hierarchy includes 235 source files encompassing 53,436 lines of code (including comments). Each compilation was verified by comparing the error messages[2] and by checksumming the generated binaries. Each virtual machine passed the test; the timing results are shown in Table 6.4. We give average user and system execution times, in seconds; as provided by the Unix `time` user command. All runs represent at least 99% system utilization, and we present the average and standard deviation over 20 runs.

JVM	Time in sec.	(σ)	Norm.
CLASSIC (v7)	17.10 + 0.17	(0.008)	0.420
CLASSIC (v8)	46.39 + 0.15	(0.146)	1.134
UNI	40.89 + 0.16	(0.170)	1.000
MSA-MI	42.24 + 0.18	(0.088)	1.034
MSA-INL	41.91 + 0.18	(0.149)	1.025
SRP-L-MI	41.92 + 0.18	(0.101)	1.026
SRP-L-INL	41.88 + 0.17	(0.140)	1.024
SRP-C-MI	41.83 + 0.17	(0.099)	1.023
SRP-C-INL	41.78 + 0.16	(0.088)	1.021

Table 6.4: Compatibility Testing and Performance
(User+System Time to Recompile `sun.tools`, in seconds; smaller is better)

We compare each of our multi-dispatch JVMs against the CLASSIC JVM — the binary distribution offered as a commercial system. One important note is that the production JDK includes an earlier version of the `javac` compiler. The Research JVM includes a v.8 compiler which accepts parametric types, similar to Sun's current proposal [48]. Therefore, we also tested the CLASSIC JVM with the v.8 compiler.

The first important result we see is that all of the multi-dispatch JVMs are faster than the CLASSIC JVM. This is an artifact of advances in compiler technology. In previous versions of this work [14, 15], we used the GCC 2.95.2 compiler; moving to the GCC 3.02 compiler provided a 3-5% performance improvement. Sun Microsystems released the production CLASSIC JDK before that compiler was available. Indeed, Sun does not indicate which compiler they use for their production releases. Hence, we take the UNI JVM as a baseline — it contains no multi-dispatch code and represents a consistent basis for comparison of the multi-dispatch JVMs.

We see that the multi-dispatch JVMs suffer a 2-3% performance penalty, with the SRP versions taking the smaller penalty. In the MSA-MI and MSA-INL tests, no multi-dispatch code is being run. Therefore, the overhead must result from other differences, including the additional class-loading checks and reduced locality of reference caused by larger data structures. We turn our attention to the overhead applied to uni-dispatch alone.

---

[2] There is one warning noting that 8 files use deprecated APIs.

Uni-Dispatch Performance

Our second quantitative experiment is to take a simple uni-dispatch loop, and execute it 10,000,000 times. In turn, 20 executions of this loop are averaged. The loop corresponds to the `AWT` kernel illustrated in Figure 2.3, but with no second dispatch. See Figure B.2 for the code. In addition to illuminating the performance of uni-dispatch, it will provide a point of comparison for double-dispatch implementations. The results are given in Table 6.5.

JVM	Time	(σ)	Norm.
CLASSIC	0.01643	(0.0002)	1.034
UNI	0.01589	(0.0003)	1.000
MSA-MI	0.01560	(0.0003)	0.981
SRP-L-MI	0.01590	(0.0004)	1.001
SRP-C-MI	0.01590	(0.0020)	1.001
MSA-INL	0.01630	(0.0003)	1.025
SRP-L-INL	0.01631	(0.0003)	1.026
SRP-C-INL	0.01631	(0.0003)	1.026

Table 6.5: Uni-Dispatch Compatibility Testing and Performance
(Call-site Dispatch Times in microseconds, smaller is better)

We see that the UNI JVM is faster than the CLASSIC one; again, we suspect the newer compiler as well as some code cleanups we performed when inserting the in-line multi-dispatch tests.[3] Again, we take UNI as our baseline. We will no longer show the CLASSIC JVM because its results are unrevealing. We cannot explain why the MSA-MI JVM is 1.9% faster than our baseline; perhaps fortuitous data placement has tweaked some increased cache efficiency. The inline JVMs labour under a 2.5% slowdown, presumably from the inline flag testing and increased cache misses due to larger structure sizes.[4]

Overall, the selection of compilers appears to have a stronger effect than our multi-dispatch extensions.

Double Dispatch Performance

Our third quantitative experiment is to take the double-dispatch kernel of the `AWT` event processing loop, and to re-write it using the Visitor pattern (the fastest of the various double dispatch techniques). In this way, every `Component.processEvent(Event)` operation becomes two uni-dispatches. In this test, both the receiver and argument virtual function tables are dispatched through, so we expect less cache effect. In each test, we double-dispatch seven different event types across seven different component types, repeated 1,000,000 times. We also repeat each entire test 20 times, and provide the average and standard deviation in Table 6.6.

---

[3] We measured the impact of those cleanups, and they resulted in a 0.03 microsecond difference per uni-dispatch.

[4] Discussion with local researchers in code generation and language implementation provided a rule of thumb that a 3% variation occur from these effects.

JVM	Time	(σ)	Norm.
UNI	0.2413	(0.0002)	1.000
MSA-MI	0.2356	(0.0004)	0.980
SRP-L-MI	0.2415	(0.0003)	1.001
SRP-C-MI	0.2416	(0.0003)	1.001
MSA-INL	0.2429	(0.0004)	1.006
SRP-L-INL	0.2455	(0.0003)	1.017
SRP-C-INL	0.2426	(0.0003)	1.006

Table 6.6: Visitor Pattern Compatibility Testing and Performance
(Call-site Dispatch Times in microseconds, smaller is better)

Other than the anomalous MSA-MI performance, we see that the relative performance gap has narrowed, confirming our suspicion that caching was boosting the UNI JVM results compared to the table-based systems. As an experiment, we compiled the UNI JVM using GCC 2.95.2, and saw a 7% performance loss per dispatch — the same dispatches took $0.2587\mu s$ each.

The multi-dispatch overhead appears to be less than 0.2% for the multi-invoker, and 2.5% for the inline testing implementations. Both of these results are overwhelmed by other factors such as compiler differences. Note that in our implementation, table-based JVMs do not construct a dispatch table until the first multi-dispatchable method is inserted.

6.2 Correctness

The second criterion consists of constructing a number of tests and ensuring that each bytecode and invocation mode correctly dispatches. There is nothing to evaluate here, but to note that a broad array of tests verifying each `invoke` bytecode were developed and executed. In addition, tests of the standard multi-dispatch cases were performed, including careful attention to the special cases noted throughout this dissertation. Examples of the cases include: `null` arguments, deliberately ambiguous dispatches, deliberately incompatible return types, and array argument types. Last, voluminous debugging output was inspected to verify that type-numbering occurs in the "lazy" fashion described, as well as multi-dispatch on unresolved methods forcing the argument class-loading only when required.

The code for many of the tests can be found in Appendix C.

6.3 Multi-Dispatch Performance

The second criterion we evaluated for is performance against equivalent uni-dispatch code. As we saw in Section 2.2, multi-dispatch eliminates the need for custom-coded double dispatch. Therefore, we will begin by comparing our multi-dispatch JVMs against equivalent double dispatch code executing on uni-dispatch JVMs.

6.3.1 Multi-Dispatch Versus Double Dispatch Performance

Our key evaluation is to compare double dispatch against our various multi-dispatch JVMs and demonstrate the efficiency of multi-dispatch.

We selected the kernel of the JDK 1.2.2 `AWT` event dispatcher as the foundation for our evaluation. This is a useful and practical example, because it is used countless times every day. It forms the heart of the `AWT` and `Swing` libraries. Every operating-system event — keypress, mouse move, window exposure, etc. — generates an event that gets queued into an event list and eventually dispatched to a visual component in a window. Those components register listeners, and the actual event processing implementation, for example, `ListBox.processKeyEvent()`, executes callbacks to those listeners (the so-called "new `AWT` event model"). The crucial insight is that the top-level `processEvent()` implementation is still executed for every event, and that it double dispatches those events based on component and event types.

We selected the JDK 1.2.2 event dispatcher over the 1.3.0 event dispatcher because it executes faster under double dispatch. The JDK 1.3.0 event handler accepts one more event type, `HierarchyEvent` than Java 1.2, resulting in yet another typecase statement to be executed. In addition, the `HierarchyEvent` typecase contains a subordinate type field dispatch, similar to how `MouseEvent` is handled. As a result, the JDK 1.3.0 event handler performs approximately 12% slower than the Java 1.2.2 event loop — but our multi-dispatchers perform identically. In addition, the JDK 1.2.2 event dispatcher was used in our previous work [14, 15].

As we shall see, Sun Microsystems' implementation of the event handler is not the most efficient method. To support our claim that multi-dispatch is competitive with double dispatch, we must compare against other, more efficient versions of double dispatch. Therefore, we rewrote the kernel of the JDK 1.2.2 `AWT` event dispatcher `Component.processEvent(AWTEvent)` in a variety of double-dispatch formats. They are described below, and the base code can be found in Appendix B.

kernel (K): This is the basic kernel as it exists in Sun Microsystems' `java.awt` 1.2.2 library.

visitor pattern (V): This is the generally accepted design pattern of using uni-dispatch to identify a receiver, then re-uni-dispatch using the argument as another receiver. This second method has the exact types of both the receiver and argument, and can execute type-specific code. This subsumes the strategy pattern — indeed a visitor can be considered as a strategy with only one method, and hence not needing the separate callback object.

typecase (TC): The system uni-dispatches base on the receiver type into a application-programmer written dispatch method that tests the type of the argument via `if-else if-else` using `instanceof` to perform the type discrimination. In this case, once both the receiver and argument types are known, the programmer-written dispatch method invokes another method containing the type-specific code.

inlined typecase (TC-I): This is identical to the previous case, except that the type-specific code is inlined into the custom programmer-written dispatch method.

typefield via accessor (TFA): The system uni-dispatches based on the receiver type, into a custom programmer-written dispatch method that tests the type of the argument by calling a `final` accessor method that returns an integer typefield for the argument. This typefield is tested in a `switch` and once both the receiver and argument types are known, the custom programmer-written dispatch method invokes a method containing the type-specific code.

inlined typefield via accessor (TFA-I): This is identical to the previous case, except that the type-specific code is inlined into the custom programmer-written dispatch method.

typefield direct (TFD): The system uni-dispatches based on the receiver type into a custom programmer-written dispatcher that tests the type of the argument by reading a `final` instance field from the argument. That typefield is an integer that is tested in a `switch` and once both the receiver and argument types are known, the custom programmer-written dispatch method invokes a method containing the type-specific code.

inlined typefield direct (TFD-I) This is identical to the previous case, except that the type-specific code is inlined into the custom dispatcher.

Inlined Double Dispatch

The inlined examples stretch the definition of OO programming, by eliminating any possibility of refining the behaviour of a subclass without replacing the entire method. As our example code shows in Appendix B, the dispatcher must be replicated and customized for each receiver class. There is no code reuse, beyond cut-and-paste; inheritance plays no role. This hardly qualifies as object-oriented. But, it is legitimate Java code, and we have seen examples where inlined dispatchers are used. Hence, we include them in our analysis.

We compare these against the kernel of the `AWT` dispatcher found in Sun Microsystems' implementation of `Component.processEvent(AWTEvent)` from the JDK 1.2.2. That kernel is a combination of a typecase for most events, except `MouseEvent` which has a second layer of type field dispatch.[5] None of the type-specific code is inlined — a second method, for example `ListBox.processFocusEvent(FocusEvent)`, is invoked.

For multi-dispatch, we provide two implementations: the simple multi-dispatch seen in Figure 2.3 where only the typecase is replaced with multi-dispatch, and a more complete solution where the previously unified `MouseEvent` class is divided into different classes.

We provide the first implementation because it requires the least code changes: the first multi-dispatch solution does not require any new classes. This limits the code changes to removing the double dispatch implementation (`Component.processEvent(AWTEvent)`) and renaming the type-specific implementations (e.g. `ListBox.processFocusEvent(FocusEvent)` becomes `ListBox.processEvent(FocusEvent)`. The secondary double-dispatch, based upon the `AWTEvent.id` field for `MouseEvents` remains however, becoming a `Component.processEvent(MouseEvent)` method.

The fully multi-dispatch solution applies our multi-dispatch JVMs more broadly, by introducing new types, for example `MouseDraggedEvent` and `MouseMovedEvent`, for the various kinds of `MouseEvents`.[6] The type field code disappears as well, and a number of new implementations, for instance, `ListBox.processEvent(MouseDraggedEvent)` and `ListBox.processEvent(MouseMovedEvent)`. This requires more code changes, but entirely removes double dispatch from event processing and better represents the performance of our multi-dispatch implementations.

In each test, we double dispatch seven different event types across seven different component types, repeated 1,000,000 times. We also repeat each entire test 20 times. Before drawing comparisons, we provide the results for executing the two multi-dispatch implementations on each multi-dispatch JVM. This will give us an indication of the relative performance of the various multi-dispatch JVMs. The results are shown in Table 6.7.

	Simple MD		Fully MD	
JVM	Time	(σ)	Time	(σ)
MSA-MI	1.4880	(0.0008)	1.4135	(0.0006)
SRP-L-MI	0.5745	(0.0006)	0.3791	(0.0012)
SRP-C-MI	0.4105	(0.0013)	0.2522	(0.0002)
MSA-INL	1.4251	(0.0022)	1.3848	(0.0002)
SRP-L-INL	0.5236	(0.0003)	0.3507	(0.0002)
SRP-C-INL	0.3686	(0.0004)	0.2240	(0.0013)

Table 6.7: Multi-Dispatch Performance
(Call-site Dispatch Times in microseconds, smaller is better)

Our reference implementations, MSA-MI and MSA-INL perform poorly compared to the other implementations. This is expected — the MSA algorithm is a reference platform intended to implement the exact semantics given by the Java static multi-dispatch algorithm.

---

[5] The JDK 1.3.0 adds another event type, `HierarchyEvent` with its own second layer of type field dispatch. We use the JDK 1.2.2 kernel because it was the basis for our previous work, and because the JDK 1.2.2 kernel performs better under double dispatch.

[6] The JDK 1.3.0 event loop would require additional `HierarchyEvent` subclasses to be created as well.

We see approximately a 0.035μs performance difference between the inline and multi-invoker versions on the event kernel. This absolute difference indicates the cost of having the multi-invoker re-construct the `invoke` bytecode.

In addition, the fully multi-dispatch implementation is faster than the simple version. With the MSA algorithm, the performance difference is slight. This is as expected: the MSA algorithm is very slow, so the added overhead of re-dispatching (via type fields) one out of every seven method invocations is not as large an impact. The tuned SRP dispatcher shows considerable benefit from eliminating the subordinate double dispatch. The cost of a type field-based double dispatch, even for only one of every seven events, adds a 50% overhead to the dispatch.

MSA Implementation Performance

Using these results, we present comparisons against the various double-dispatch implementations on the baseline UNI JVM. We report the UNI timing, and provide the speedups gained by the two kernel implementations, where UNI is normalized to 1.0.

We begin with the UNI versus MSA-based JVMs, shown in Table 6.8.

	Simple MD		Fully MD	
Dispatch	MSA-MI	MSA-INL	MSA-MI	MSA-INL
TC	0.393	0.410	0.413	0.422
K	0.348	0.363	0.366	0.374
TFA	0.297	0.310	0.313	0.319
TC-I	0.270	0.282	0.285	0.291
TFD	0.204	0.213	0.215	0.219
TFA-I	0.163	0.170	0.172	0.175
V	0.162	0.169	0.171	0.174
TFD-I	0.068	0.071	0.071	0.073

Table 6.8: Double vs. Multi-Dispatch — part I: MSA
(Speedups against UNI = 1.000, larger is better)

We see that the inlined version is approximately 4% faster than the multi-invoker version. None of the MSA versions are competitive with double-dispatch; but that is not their purpose.

SRP (Simple Locking) Performance

Next we look at the results for our tuned SRP implementation, with the simple locking algorithm. The results are summarized in Table 6.9.

The tuned dispatcher performs much better, providing faster dispatch than half of the double dispatch implementations. But the 0.17μs overhead for entering and exiting the behaviour monitor (described in Section 5.2.2) is still too high.

Dispatch	Simple MD SRP-L-MI	 SRP-L-INL	Fully MD SRP-L-MI	 SRP-L-INL
TC	1.017	1.116	1.394	1.666
K	0.902	0.989	1.236	1.477
TFA	0.769	0.844	1.055	1.261
TC-I	0.700	0.769	0.960	1.148
TFD	0.529	0.580	0.725	0.866
TFA-I	0.422	0.463	0.578	0.691
V	0.420	0.461	0.576	0.688
TFD-I	0.175	0.192	0.240	0.287

Table 6.9: Double vs. Multi-Dispatch — part II: SRP-L
(Speedups against UNI = 1.000, larger is better)

SRP (CREW Locking) Performance

Next we look at the results for our tuned SRP implementation, with the concurrent-read–exclusive-write algorithm. The results are summarized in Table 6.10.

Dispatch	Simple MD SRP-C-MI	 SRP-C-INL	Fully MD SRP-C-MI	 SRP-C-INL
TC	1.541	1.585	2.317	2.609
K	1.366	1.405	2.053	2.313
TFA	1.166	1.199	1.753	1.974
TC-I	1.062	1.092	1.595	1.797
TFD	0.801	0.824	1.204	1.356
TFA-I	0.640	0.658	0.961	1.083
V	0.636	0.654	0.957	1.077
TFD-I	0.265	0.273	0.399	0.449

Table 6.10: Double vs. Multi-Dispatch — part III: SRP-C
(Speedups against UNI = 1.000, larger is better)

These results show that our tuned SRP dispatcher, with high-performance concurrency control, can out-perform user-level double dispatch. For example, our tuned SRP-C-INL dispatcher locates the correct event handler in 0.2240μs, whereas typecases (TC) on the UNI JVM require 0.5179μs, more than 2.6 times as long. The only double dispatcher we could not best was the inlined direct-access type-fields. As we commented above, that double dispatcher is not OO code. It chooses to re-implement types using its own numeric scheme. It gains code reuse or extensibility from inheritance. The TFD-I dispatcher is brittle and difficult to maintain: adding a new component or event requires modification to every component and event.

Our tuned SRP dispatcher marginally out-performs the fastest OO double dispatcher, the visitor pattern. The Visitor pattern (V) consumes 0.2413μs to locate the event handler, just 1.077 times as long as our multi-dispatch JVM. Our tuned SRP implementation provides lower latency dispatch than another non-OO double dispatcher, TFA-I. Specifically, TFA-I requires 0.2424μs, 1.083 times as long as SRP-C-INL. It provides more than double the performance of Sun Microsystems' default kernel and the typecase double-dispatcher used in other source-translation versions of multi-dispatch Java [11, 12].

6.3.2 Arity Effects

Our final micro-benchmark explores the time penalties as the number of dispatchable arguments and applicable methods grow. To do this, we built a simple hierarchy of five classes (one root class `A`, with three subclasses `B`, `C`, and `D`, and finally class `E` as a subclass of `C`) and constructed methods of different arities against that hierarchy. We defined the following methods:

- classes `A`, `B`, `C`, `D`, and `E` contain unary methods `R.m()` (where R represents the receiver argument class).
- classes `A`, `B`, `C`, `D`, and `E` also implement five binary methods, `R.m(X)` where X can be any of `A`, `B`, `C`, `D`, or `E`.
- classes `A`, `B`, `C`, `D`, and `E` implement 25 ternary methods, `R.m(X,Y)` where X and Y can be any of `A`, `B`, `C`, `D`, or `E`.
- classes `A`, `B`, `C`, `D`, and `E` implement 125 quaternary methods, `R.m(X,Y,Z)` where X, Y, and Z can be any of `A`, `B`, `C`, `D`, or `E`.

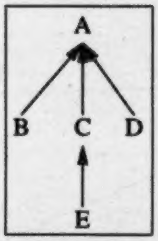


Figure 6.1: Arity Effects Test Hierarchy

MSA looks at one fewer dispatchable argument than the table-based techniques because the receiver argument has already been dispatched by the JVM. For instance, given a unary method, MSA makes no widening conversions for dispatchable arguments. A binary method requires MSA to check only one widening conversion. The table-based techniques dispatch on all arguments and gain no benefit from the dispatch done by the JVM.

We invoke 1,000,000 methods for each arity. This means that each of the unary methods is executed 200,000 times. However, each of the quaternary methods is executed only 1,600 times. After computing the loop overhead via an empty loop, we determine the elapsed time to millisecond accuracy and determine the time taken for each dispatch. Our results are shown in Figure 6.2.

We can evaluate the arity effects in the uni-dispatch case by coding a third level of double dispatch. Already the overhead of constructing a third activation record exceeds the dispatch time of our tuned SRP implementation. Also, our SRP implementations suffer only

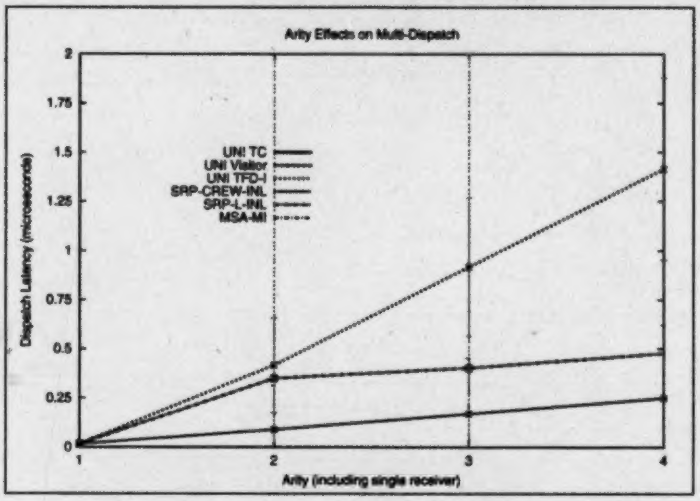


Figure 6.2: Impact of Arity on Dispatch Latency

linear growth in time-penalties as arity increases, whereas MSA suffers exponential effects. Our results are shown in Table 6.11.

Receivers	Unary	Binary	Ternary	Quaternary
Example	`R.m()`	`R.m(X)`	`R.m(X,Y)`	`R.m(X,Y,Z)`
JVM	Time	Time	Time	Time
TC	0.0159	0.4174	0.9166	1.4158
V	0.0159	0.2413	0.3520	0.4627
SRP-L-INL	0.0159	0.3507	0.4034	0.4781
SRP-C-INL	0.0159	0.2240	0.2831	0.3722
MSA-INL	0.0156	2.4700	9.8807	—

Table 6.11: Arity Effects Performance
(Call-site Dispatch Times in microseconds, smaller is better)

MSA suffers from exponential effects, because it linearly searches through an exponentially growing number of implementations. Typecase-base dispatch grows linearly with the number of arguments, but with a relatively expensive test. What is not clear in this table is that the testing costs also grow linearly with the number of types involved, because the typecase is a sequence of `if ... else if ... else ...` constructs.[7] Double dispatching with the Visitor pattern is linear in the number of arguments, and constant in the number of types. Our tuned SRP dispatchers are linear in the number of arguments, and constant in the number of types.

---

[7] Even worse, because the order of the tests is dictated by the type hierarchy, one cannot optimize the typecase for common occurrences.

6.3.3 `Swing` and `AWT`

For our application-level tests, we modified `Swing`, the second-generation GUI library bundled with Java 1.3.0, to use multi-dispatch. As expected, `Swing` is a double-dispatch-intensive library. We also converted `AWT` because `Swing` depends heavily on `AWT` to dispatch the events into top-level `Swing` components.

In the final benchmark, `Swing`, we report execution times for a synthetic application that creates a number of components and inserts 200,000 events into the event queue.

	Uni-Swing	Multi-Swing	
Stage	Methods	Uni-Methods	Multi-methods
warm-up	901,938	901,795	160 (0.02%)
event loop	32,543,684	27,807,327	2,350,172 (7.7%)

Table 6.12: `Swing` Application Method Invocations

We modified 11% (92 out of 846) of the classes in the `AWT` and `Swing` hierarchies. We eliminated 171 decision points, but needed to insert 123 new methods to replace existing double-dispatch code sections. Within the modified classes, we removed 5% of the conditionals and reduced the average number of choice points per method from 3.8 to 2.0. This measure, related to the *cyclomatic complexity number* [34], encompasses the design and structural complexity of the code. Our two-fold reduction illustrates the value of multi-dispatch in reducing code complexity.

In all, 57 classes were added, all of them new event types to replace those previously recognized only by a special type id (as in the `AWT` examples described previously). Our multi-dispatch libraries are a drop-in replacement that executes a total of 7.7% fewer method invocations and gives indistinguishable performance with applications such as `SwingSet`. Many demonstration programs are interactive, so precise timing is impossible. Therefore, we constructed a sample application which operates unattended. In our sample application, we found that the number of multi-dispatches executed almost exactly equaled the total reduction in method invocations. This suggests that every multi-dispatch replaced a double dispatch in the original `Swing` and `AWT` libraries.

We verified the operation of the entire unmodified `SwingSet` application with our replacement libraries. Finally to measure performance, we timed a simple `Swing` application that handles 200,000 `AWTEvents` of different types. The timing results from 10 iterations are given in Table 6.13.

Despite our high performance multi-dispatcher, we do not see significant gains in performance. This arises for several reasons. First, not all arguments need to be dispatched upon; hence custom dispatchers can ignore some no-variant arguments and thereby reduce latency. Our dispatcher does not do this (yet). Second, there is a small overhead with our SRP-C-INL dispatcher; we measured it at about 2.5% previously. Third, class loading and behaviour

Dispatch JVM	Uni-Swing Time	Uni-Swing (σ)	Multi-Swing Time	Multi-Swing (σ)
UNI	14.85	(0.35)	—	—
MSA-MI	14.99	(0.28)	37.44	(0.62)
SRP-L-MI	15.00	(0.24)	17.03	(0.21)
SRP-C-MI	15.00	(0.17)	14.92	(0.24)
MSA-INL	15.02	(0.19)	38.01	(0.17)
SRP-L-INL	15.02	(0.26)	16.74	(0.20)
SRP-C-INL	15.02	(0.24)	14.69	(0.11)

Table 6.13: `Swing` Application Execution Time
(Application loop times in seconds, smaller is better)

construction add to the overhead of the multi-dispatch solution. Finally, multi-dispatch improves the performance of a very small portion of the entire application. Multi-dispatch constitutes 7% of the method invocations only. Roughly estimating, with 2,300,000 multi-dispatches, saving at most 0.3μs, roughly the difference between Sun Microsystems' kernel and our fastest dispatcher, we would expect a gain of at most 0.7 seconds. We see only a fraction of that.

However, the `Swing` and `AWT` conversion also demonstrates the robustness of our approach. We needed to support multi-dispatch on instance and static methods. `Swing` and `AWT` expect to dispatch differently on `Object` and array types. In modifying the libraries, we found numerous opportunities to apply multi-dispatch to private, protected, and super method invocations. In addition, several multi-methods required the JVM to accept co-variant return types from multi-methods. Array arguments performed quite well; by monitoring the creation of fake array classes during the execution of the various demo programs we saw that no extra array classes were created compared to the original JVM. All of these features are required for a mainstream programming language.

6.4 Summary

Our evaluation has shown that multi-dispatch is compatible with the existing Java language. Our marker interfaces allow programmers to target multi-dispatch to the places where it is effective, without diminishing the performance of the remaining uni-dispatch code. The overhead is shown to be less than 2.5%, less than the effects of other variables, such as compiler technology. Existing uni-dispatch code remains syntactically and semantically unchanged.

The tuned SRP dispatcher provides high-performance dispatch, rivaling the most efficient OO double-dispatch techniques. Compared to simple, yet commonly used, double dispatch techniques, SRP can dispatch more than twice as quickly. Our technique is linear in the number of arguments, so it scales well to higher arity dispatch.

Despite all its promise, multi-dispatch is not a performance panacea. Our tests with a "classic" double dispatch application, event-driven graphical user interfaces, suggests that double dispatch comprises less than 8% of the dispatches in out test application. Furthermore, those dispatches comprise only a small fraction of the operations that make up the entire application. Therefore, although our multi-dispatch technique is efficient, it does not provide dramatic overall performance improvements. The real benefits come from eliminating the plethora of application-level dispatchers. Our results demonstrate that achieving the software engineering benefits of multi-dispatch does not compromise efficiency.

Chapter 7

Discussion

We conclude this dissertation with some remarks about other work in the area of multi-dispatch and Java, some future projects, and a short summary of this dissertation.

7.1 Related Work

Others have attempted to add multi-dispatch to Java through language preprocessors. Boyland and Castagna [4] provide an additional keyword *parasite* to mark methods which should have multi-dispatch properties. They effectively translate these methods into equivalent double-dispatch Java code. By translating directly into compiled code, they apply a lexical priority to avoid the thorny issue of ambiguous methods. Unfortunately, the parasitic method selection process is a sequence of several typecase dispatches to search over a potentially exponential tree of overriding methods.

Another recent development is *MultiJava* [12, 11]. There, the authors extend the Java language with additional syntax to support open classes and multi-dispatch. The MultiJava compiler emits double-dispatch typecase bytecodes for invocations of the open-class methods and multi-methods. The emitted bytecode is accepted by standard JVMs, but suffers a substantial overhead from interpreting slow `instanceof` bytecodes. Clifton's results show overhead of 1.6 times the cost of visitor pattern double-dispatch — but this is on an application with only three types.[1] Further, Clifton's multi-dispatch can only apply to methods defined using the open-class syntax and only within program text that explicitly imports the open-class definitions.

Multi-dispatch is not automatically available to subclasses in Multi-Java, violating the expected inheritance properties, and leading to unexpected results. If subclasses wish to further specialize the multi-methods, additional open-class definitions are required. Compilation of these further open-subclasses may result in multiple layers of typecase double-dispatch. Internally, MultiJava inlines the multi-method bodies into a static method in a separate anchor class — this means that the multi-methods disappear from the binary code and become invisible to the reflective subsystem in Java.

---

[1] Recall that typecase performance is linear in the number of types tested

Another project is the *Java Multi-Method Framework* [17, 18] which applies Java reflection to determine the collection of methods and classes which may be multi-dispatched. Their system provides a Java API which allows Java code to register methods into a multi-dispatch structure, and to perform a multi-dispatch. By providing programmers with this level of control, they offer very fine-grained dispatch control. Unfortunately, their approach suffers from a few practical limitations. First, their dispatch performance is approximately 40 times slower than our optimized JVM approach. Second, they class-load all argument types when a multi-method is registered — this reduces the efficiency gains realized by the JVM's lazy class-loading technique. Last, they do not appear to handle array arguments at all; this is most likely a result of the limited support for arrays in the Java reflection API.

In comparison, Gupta [23] discusses an MSA version of the Forax approach. Gupta does not give an implementation, but a simple one is available online at [36]. This system has neither the elegant API that Forax offers nor the higher performance. Performance numbers are not available. MacDonald [33] also incorporates some elements of MSA multi-dispatch into his `MethodThread` class, but that is not its primary focus.

The language extension and preprocessor approach has other limitations. First, existing tools do not support the extensions; for example, debuggers do not elide the automatically generated double-dispatch routines. Second, instance methods appear to only take arguments that are objects, which is too limiting. Our experience with `Swing` shows that existing programs often double dispatch on literal `null` and array arguments and pass primitive types as arguments; multi-methods need to support these non-object types. Third, preprocessors limit code reuse and extensibility; adding multi-methods to an existing behaviour requires either access to the original source code or additional double-dispatch layers.

In contrast, extending the Java Virtual Machine to support multi-dispatch directly opens up new opportunities. Clearly, the extended languages described above can avoid generating slow and complex dispatch code by taking advantage of native multi-dispatch in the JVM iself. In addition, there are now many language implementations that rely upon the JVM as the runtime system — they compile to `.class` files. Tolksdorf [49] currently lists 160 different languages that compile to the JVM specification, including object-oriented languages such as as Eiffel and Smalltalk. Our compatible multi-dispatch JVM enables extending those languages to multi-dispatch as well.

Chatterton [10] examines two different multi-dispatch techniques in mainstream languages: C++ and Java. First, he considers providing a specialized dispatcher class. Each class that participates as a method receiver must register itself with the dispatcher. To relieve the programmer of this repetitive coding process, he provides a preprocessor that rewrites the Java source to include the appropriate calls. Each method, marked with the keyword *multi*, is also expanded by the preprocessor into many individual methods, one for each combination of classes (and superclasses). A method invocation is replaced by a call to the dispatcher which searches via reflection for an exact match. That method is then invoked. This system suffers from exponential blowup of methods.

Chatterton's second approach examines the performance of various double dispatch enhancements. He provides a modified C++ preprocessor which analyses the entire Java program. It can build a number of different double-dispatch structures, including cascaded and nested `if ... else-if ... else` statements, inline `switch` statements, and simple two-dimensional tables. Again, he expands every possible argument-type combination in order to apply fast equality tests rather than slow subtype checks. A significant restriction is that full-program analysis is required. This defeats the ability to use existing libraries and diminishes Java's dynamic class loading benefits.

One interesting language for multi-dispatch is Leavens and Millstein's *Tuple* [29]. They describe a language "similar in spirit to C++ and Java" that permits the programmer to specify at each call-site the individual arguments that will be considered for multi-dispatch. Their paper does not describe an implementation; it appears to be a model of potential syntax and semantics only. A future project might be to implement his syntax specifically into the Java environment. In particular, a simple syntax extension would allow `super` method invocations on arbitrary multi-dispatch arguments; we discuss this more detail as a future activity.

7.2 Future Work

There are several areas where our work can be extended and applied. We begin by examining some potential variations on arguments and algorithm optimization. Then we look at broader applications of multi-dispatch.

7.2.1 Just-In-Time Compilers

We have begun work on integrating our multi-dispatchers into the OpenJIT [35] Just-In-Time (JIT) compiler framework. This is actually a straightforward task, because the structure of the OpenJIT framework closely matches that of the JVM itself. In particular, invokers are still operational; and the invoke bytecodes become inlined calls to stub routines which perform the method lookup.

We have the multi-invoker working with the OpenJIT system at this time, giving comparable performance results to the interpreted JVM. Our dispatcher operates at the identical speed, and OpenJIT uni-dispatches marginally slower than the optimized assembly language interpreter loop. Dispatch-level benchmarks show multi-dispatch still exceeds double-dispatch with OpenJIT. Preliminary application-level benchmarks show that multi-dispatch Java retains the dramatic performance gains from the JITing the many other instructions in the program. As expected multi-dispatch itself shows only a slight performance gain over double-dispatch. Our initial results show that one does not have to sacrifice JIT performance in order to gain the benefits of use multi-dispatch.

The multi-invoker implementation was simple: OpenJIT operates by replacing the invoker with another which prepares for and executes a processor-specific compiled version of the method. Our changes involve retaining the multi-invoker for multi-dispatch methods, and having OpenJIT replace the cached (original) invoker instead.

The inlined-test version for OpenJIT is incomplete. We have only two bytecodes working with multi-dispatch at this time: `IV` and `INV`. The latter tweaked an implementation issue with the JVM where `invokesuperquick` was implemented as a lossy bytecode, but not treated as such by OpenJIT. We do not have performance results at this time, but anticipate that we will see the same effects as the multi-invoker implemenmtation.

7.2.2 Additional Optimizations

Our algorithm is quite efficient, but it could be improved in several ways. First, we do not explicitly check for the trivial case where a behaviour has only one implementation. Although this is a rare occurrence, some space savings can be achieved by recognizing that no multi-dispatch needs to be done. The only implementation must apply, because of static type-checking, and it is trivially the most specific. The potential difficulty is that every multi-dispatch would need to check for this case, adding a slight performance penalty to the most common multi-dispatch case. It is not clear what the optimal choice is.

A second optimization comes from considering the case where the number of dispatched slots is one. This never occurs for virtual multi-methods, but appears common in static and constructor multi-methods with one dispatchable argument. In this case, the process of peeking at the argument stack could be optimized with custom code (just as Sun Microsystems did with the common invokers), and `bits` could incorporate the `overrides` directly. This would eliminate the computation of `bits[row]` and remove the `mbits & overrides` computation. In essence, static and constructor behaviours with only one dispatchable argument would reduce to a virtual function table indexed by the dispatchable argument.

A third optimization derives from recognizing that the higher arity multi-methods frequently have some no-variant dispatchable arguments. For example, the `Component.addPropertyChangeListener(String,PropertyChangeListener)`, `Component.removePropertyChangeListener(String,PropertyChangeListener)`, `Component.firePropertyChange(String,Object,Object)` do not vary their first argument type: it is always `String` — a `final` class. It is pointless to dispatch on this argument position. Indeed, this optimization could be developed into a full-blown behaviour optimizer. As methods are added (or removed as classes are unloaded), the behaviour can change shape to examine the minimum number of arguments. Rows for dispatchable arguments might be added, deleted, or replaced as dictated by the available implementations.

A fourth optimization is to provide a compression technique over the rows or columns the behaviour tables. Colouring techniques, as described in [2] can be applied over the rows of a single behaviour, or between rows of multiple behaviours. Alternately, one can colour over columns, i.e. type numbers, in recognizing that multi-methods frequently vary over disjoint trees of types. In this case, the dispatch bitfields for one or the other of the disjoint type-number sets are always null, and can be eliminated by reusing the same type numbers for classes in both sets. The type-numbering algorithm becomes more complex, but table sizes can be dramatically reduced.

As a fifth optimization, it is possible to memoize the verification of return types, and avoid the expense of a subtype test at each dispatch. In the general case for a behaviour with with N methods, this would require an N-bit bitfield for each method. This is a result of the freedom that Java offers to overloaded methods. Despite the restrictive novariant return types enforced for overriding methods, Java enforces no restrictions on the return values for overriding methods. Hence, it is possible for any overriding method to provide a return value that a subtype, a supertype, or unrelated to the return value of any of the other implementations comprising a behaviour. Providing a flag for each combination of uni-dispatch `i` and multi-dispatch result method `m`, we can eliminate repeating the sub-type test in the common case. If bit `i` of `returntype[m]` is not set, the existing subtype test is be performed. If the test shows that the multi-dispatch implementation returns a value that is a subtype of that returned by the uni-dispatch implementation, bit `i` of method `returntype[m]` is set. If the subtype-test fails, then an `IllegalReturnTypeError` exception is thrown. In the error case case, we accept the overhead a duplicated subtype test. We determined a bound on the performance benefit possible for this optimization by removing the subtype test entirely; the beneift would be at most than $0.022\mu s$, or approximately 10% for our high-performance SRP dispatcher. But this would need to be reduced somewhat by the overhead of storing the behaviour position number into each offset and by the overhead of the bitfield operations.

7.2.3 Explicit `null` Parameters

Consider the behaviour shown in Figure 5.6, or another based on the "multi-dispatch diamond" with the `null` type completing the shape (Figure 7.1). The type lattices are very complex, but it is easily seen that some conflict methods can never be written in Java. For Figure 5.6, the ambiguity is between `Super.smethod(A)` and `Super.smethod(A[])`. The only type which is a sub-type of each argument is `null`. We would like to define a method `Super.smethod(null)` which would provide a most specific applicable implementation between these two.

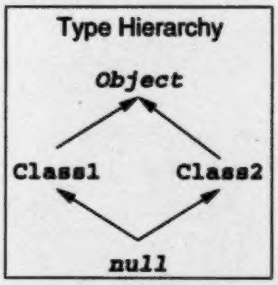


Figure 7.1: Conflict Method Not Definable

For Figure 7.1, the pairs `Class1.method(Class1)` and `Class1.method(Class2)`, `Class2.method(Class1)` and `Class2.method(Class2)` each require a conflict method with an argument of explicitly `null` type. The Java language does not supply a way to write an explicit `null` in a parameter list, so these methods cannot be defined. It would be a simple extension to permit the syntax `Class1.method(null)` and `Class2.method(null)`.

The other conflict methods, `null.method(Class1)` and `null.method(Class2)` require more than syntax changes. The structure of Java class files places all of the methods with a given receiver type into a single class. But multi-methods accepting a `null` receiver reside in all classfiles — because `null` is a sub-type of every class. This means that without every class loaded, `null` receivers cannot be permitted. An alternate approach, to supply a classfile for the `null` type, is unworkable: every new class and multi-method would typically engender a change to that `null.java` file. Some alternate approach, perhaps grouping multi-methods by behaviour, might work.

These `null` receivers and arguments are entirely plausible, in fact our behaviour structure and dispatch algorithm fully supports them. It is Java that cannot provide support them. Lazy class loading eliminates the ability to catalogue all potential classes and methods.

7.2.4 super On Non-Receiver Arguments

Uni-dispatch Java provides a way to escape from the rigid dynamic dispatch operation through "super" calls. These super calls only apply to the argument that uni-dispatch Java dynamically dispatches on, namely the receiver. For other arguments, uni-dispatch Java allows static casts to control the static multi-dispatch operation. It may be desirable to provide some similar escape mechanism for the additional arguments that multi-dispatch dynamically dispatches on. Although we did not encounter a situation where this was required, supporting that operation is possible. First, some syntax changes would be required in order to denote the dispatchable arguments that are now to be "super"ed. Second, the system would need to rewrite the appropriate type-numbers into the array given to the dispatcher (just as we do for `SelectSuperMultiMethod()`).

7.2.5 Tuple Syntax

The *tuple* syntax [29] offers intriguing possibilities. With it, we could control which arguments are multi-dispatched and which are not (currently all object arguments are multi-dispatched). Instead of the system needing to look for no-variant arguments, the programmer could denote them directly. The `PropertyListener` code above might translate into code resembling Figure 7.2. That tuple syntax could also support the extended "super" idea as well, as illustrated.

```
Component c = new Button();
String p = "MyProperty";
Listener l = new SubPropertyChangeListerer() { ...};
// dispatchable args in tuple left of ".", non-dispatchable args in "()"
<c, l>.addPropertyChangeListener(s);
<c, l>.removePropertyChangeListener(s);
// "super" can apply to any dispatchable argument
<super(c), l>.addPropertyChangeListener(s);
<c, super(l)>.removePropertyChangeListener(s);
<super(c), super(l)>.removePropertyChangeListener(s);
```

Figure 7.2: Tuple-like Syntax

7.2.6 Parametric Polymorphism

One of the interesting challenges of this work was to fully support the type relationships found in Java. One type relationship, the induced hierarchy generated by arrays, is of particular interest. It represents a parametric type, *Array* parametrized over any primitive or object type (including arrays themselves). In order to properly multi-dispatch code similar to that of Figure 7.3, which accepts different kinds of single elements and different kinds of arrays, parametric polymorphism must be supported.

Parametric polymorphism by type-erasure [13] is insufficient. Some "methods" of *Array* are inherited, such as *anArray.elementAt(i)* (written as `anArray[i]`), but the types `int[]` and `TreePath[]` are distinct. Type erasure must maintain both as separate types; they cannot be erased back to their common ancestor *Array*. We see this in the fact that we must assign different type-numbers to these two argument types. Extending Java to support user-defined parametric types, as proposed by Sun [48], will require a similar effort to that for supporting array types. A datatype representation of the type-structure [28] perhaps as simple as a linear sequence of argument types might suffice. For example, a parameter of type `HashTable<String,List<PhoneNumber>>` might end up as a sequence of four types, `HashTable`, `String`, `List`, `PhoneNumber` where only the first one has an argument value, the remaining are used for multi-dispatch only.

```
package javax.swing;
class JTree {
  ...
  void addSelectionPath(TreePath path)       { ... }
  void addSelectionPaths(TreePath[] paths)   { ... }
  void addSelectionRow(int row)              { ... }
  void addSelectionRows(int[] rows)          { ... }
}
```

(a) Methods Overloaded On Arrays

```
package javax.swing;
class JTree {
  ...
  void addSelection(TreePath path)       { ... }
  void addSelection(TreePath[] paths)    { ... }
  void addSelection(int row)             { ... }
  void addSelection(int[] rows)          { ... }
}
```

(b) Multi-Dispatch On Arrays

Figure 7.3: Array-based Parametric Polymorphism in Multi-Dispatch Java

7.2.7 Other Dispatchable Properties — Security

With a high-performance multi-dispatch algorithm, we can turn attention to other useful properties to dispatch on. One obvious choice is the type of the currently executing method. We can implement fine-grained and dynamic security.

Imagine we have a class `somePackage.Foo` and some side-effecting operation, `op`. We want to ensure that every `Foo` object remains in a consistent state, but we have varying levels of trust of the various classes which might invoke `op`. Calls from within `Foo` presumably need little validation. Calls from within package `somePackage` need some checking, but calls from outside `somePackage` need to be rigorously screened.

Current Java provides `public`, `protected` and `private` to support this kind of security. In particular, we would have `private` methods to perform the actual operations and other methods code within `Foo` might call those directly. We might supply some `protected` methods that perform cursory checks, then invoke the `private` methods. And finally, we would have `public` methods that perform rigorous and time-consuming checks before calling the `private` implementations.

Because of name-space conflicts, these methods cannot all have the same name. We end up with a kludge similar to Figure 7.4(a).

The code in Figure 7.4(a) has several problems, particularly with code refactoring, a prevalent technique in the *extreme programming* methodology [3]. Imagine a programmer moved `doIt1` into class `Bar` — then he would need to locate and replace every call to `realOp()` with a call to `pkgOp()` — fortunately the compiler will flag any omissions as errors. Moving `doIt2` into class `Foo` should have the opposite change made. If not, performance suffers, and

```
package MyPackage;
class Foo {
    private realOp(...)  { ...}
    protected pkgOp(...)  {
        if (simpleChecks(...))  { realOp(...); }
    }
    public op(...)  {
        if (manyChecks(...))  { realOp(...); }
    }
    doIt1(...)  { ...; realOp(...); }
}
class Bar extends Foo { // in MyPackage
    doIt2(...)  { ...; pkgOp(...); }
}
// ------------------------------------------
class Outside { // in some other package
    doIt3(...)  { ...; op(...); }
}
```

(a) Static Security Example

```
package MyPackage;
class Foo {
    private op(...)  { ...}
    protected op(...)  {
        if (simpleChecks(...))  { op(...); }
    }
    public op(...)  {
        if (manyChecks(...))  { op(...); }
    }
    doIt1(...)  { ...; op(...); }
}
class Bar extends Foo { // in MyPackage
    doIt2(...)  { ...; op(...); }
}
// ------------------------------------------
class Outside { // in some other package
    doIt3(...)  { ...; op(...); }
}
```

(b) Dynamic Security Example

Figure 7.4: Security and Multi-Dispatch

the compiler never warns the programmer. The same problem with search-and-replace work arises if `doIt3` migrates into `MyPackage` or `Foo`.

The example illustrates that mixing visibility with method name-spaces can lead to extra code maintenance and frequent inefficiency. However, multi-dispatch can always take another argument, the ***host-class*** (the class of the currently executing method) and use it as another dispatchable argument. In this way, we could select the most-specific applicable `op` implementation, where `private` is more specific than `protected` which, in turn, is more specific than `public`. Code refactoring requires no method renaming; the dispatcher automatically selects the appropriate implementation.

Alternate syntax for visibility could allow different levels of trust, or different operations entirely, based upon the requesting object. This could yield capabilities similar to the `friend` modifier in C++, but more sophisticated classifications are possible. Perhaps a `publicTerminal` object might not obtain access to private income-tax information, but a `taxCollector` object might. Verifying the correct operation no longer relies on auditing (potentially many) user-coded dispatchers, but proving a single JVM-provided multi-dispatcher.

7.2.8 Fully Multi-Dispatch Java

We have spent some time considering converting the JVM to make every method invocation be multi-dispatch. Although technically feasible, this appears difficult to do with the Research JVM. Internally, it makes many assumptions about the uni-dispatch nature of the core system classes, `Object`, `Class`, `ClassLoader`, and `String`, etc., especially in the construction of the *initial braid*. Further, multi-dispatch does add significant performance penalties — it is competitive with double dispatch, but not with uni-dispatch.

Alternately, using multi-dispatch exclusively would reduce some of the complexity within the JVM. In particular, as a result of Saraswat's note about type-safety in the original Java

1.1 system [37], Sun Micrososytems introduced a complex scheme of classloader constraints into Java 1.2 to patch the holes [30]. As Saraswat shows, multi-dispatch, implemented throughout the system, can ensure JVM type-safety directly. As an aside, we could not relax the classloader constraints system with our implementation of multi-dispatch because not all dispatches are multi-dispatches.

7.3 Type-less Programming

Stroustrup [45] describes the container problem and notes that "[Stroustrup,1982b] presents this ugly code for retrieving an object from a table and using it based on a type field ... Much of the effort in C with Classes and C++ has been to ensure that the programmer needn't write such code." An original motivator for C++ was to remove needless duplication of code and eliminate the common C practise of programmer-discriminated types by tagged unions. To this end, C++ was only a limited success. The programmer was relieved of writing his own quirky dispatcher for one argument, the receiver, only. Other arguments still needed to be distinguished to provide type-specific actions. As a result, the programmer still had to write his own dispatchers — just not as many. Type annotations proliferated, being found in one form or another, in double dispatchers everywhere in the application.

The code still retained a detailed knowledge of the type hierarchy. Typecases needed to be ordered from most specific to least specific. Type fields still encoded the variety and relationships among the types involved in the program. This implicit and pervasive type knowledge mades the code fragile and inflexible. The one exception, the visitor pattern avoided encoding this type knowledge, but at what cost? It requires a dispatch implementation *in each receiver class and subclass* to reverse the receiver and argument.

Stroustrup [45] further notes that multi-dispatch was considered and discarded for C++ on two grounds: other techniques (double dispatch) existed, and no-one knew how to implement it efficiently. Those other techniques are a poor substitute, adding to the limited lifespan and broad applicability of programs. As this dissertation shows, we have efficient multi-dispatch algorithms now. Indeed, these algorithms can multi-dispatch faster in production language implementations than application-level programmers can double-dispatch.

Therefore, we envision OO languages where `instanceof` and type-casts do not exist, because they are not needed. Multi-dispatch replaces `instanceof`. Type-casts serve no purpose, because all objects are treated based on their precise dynamic type. These new OO languages would contain explicit type names only for declaring classes, methods, fields, and local variables. Code would become less brittle in the face of unexpected evolution and more easily extended to new applications. The actual program statements would become *type-less* and better for it.

7.4 Closing Remarks

We have presented the design and implementation of an extended Java Virtual Machine that supports multi-dispatch. This is the first published description of how to implement arbitrary-arity, multi-dispatch in Java. In contrast to the more verbose and error-prone double-dispatch technique, currently found in the `AWT` (Figure 2.3), multi-dispatch typically reduces the amount of programmer-written code and generally improves the readability and level of abstraction of the code.

Our approach preserves both the performance and semantics of the existing dynamic uni-dispatch in Java while allowing the programmer to select dynamic multi-dispatch on a class-by-class basis without any language or compiler extensions. The changes to the JVM itself are small and highly-localized. Existing Java compilers, libraries, and programs are not affected by our JVM modifications and the programs can achieve performance comparable to the original JVM (Table 6.10).

In a series of micro-benchmarks, we showed that our prototype implementation adds negligible performance overhead to dispatch if only uni-dispatch is used (Table 6.4) and the overhead of multi-dispatch can be competitive with explicit double dispatch (Table 6.12).

We have also introduced and implemented an extension of the Java Most Specific Applicable (MSA) static multi-dispatch algorithm for dynamic multi-dispatch. In addition, we have performed the first head-to-head comparison of table-based multi-dispatch techniques implemented in a mainstream language. In particular, we implemented Single Receiver Projections (SRP). Overall, our tuned SRP implementation performs as well (or better) than programmer-written double dispatch without the complexity, errors, and maintenance costs associated with that code.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages*, 13(2):237–268, April 1991.

[2] Pascal André and Jean-Claude Royer. Optimizing method search with lookup caches and incremental coloring. In Andreas Paepcke, editor, *OOPSLA 1992 Conference Proceedings*, volume 27 of *SIGPLAN Notices*. Association for Computing Machinery, September 1992.

[3] Kent Beck. *eXtreme Programming eXplained: Embrace Change.* Addison Wesley, 1999.

[4] John Boyland and Guiseppe Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA 1997 Conference Proceedings*, volume 32 of *SIGPLAN Notices*, pages 66–76. Association for Computing Machinery, November 1997.

[5] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[6] Timothy Budd. *An Introduction to Object Oriented Programming, Second Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.

[7] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[8] Guiseppi Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3), 1995.

[9] Craig Chambers. Object-oriented multi-methods in Cecil. In Madsen O. Lehrmann, editor, *ECOOP 1992 Conference Proceedings*, number 615 in Lecture Notes in Computer Science, pages 33–56. Springer-Verlag, June 1992.

[10] David Chatterton. *Dynamic Dispatch in Existing Strongly Typed Languages.* PhD thesis, School of Computing, Monash University, Monash, Australia, 1998.

[11] Curtis Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001.

[12] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Milstein. MultiJava: Modular symmetric multiple dispatch and extensible classes for Java. In *OOPSLA 2000 Conference Proceedings*, volume 35 of *SIGPLAN Notices*. Association for Computing Machinery, October 2000.

[13] Daniel J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114(2):273–298, 1993.

[14] Christopher Dutchyn, Duane Szafron, Paul Lu, Wade Holst, and Steve Bromling. Multi-dispatch in the Java virtual machine—design and implementation. In *OOPSLA 2000 Conference Proceedings*, volume 35 of *SIGPLAN Notices*. Association for Computing Machinery, October 2000. abstract only — extended abstract is in the Companion.

[15] Christopher Dutchyn, Duane Szafron, Paul Lu, Wade Holst, and Steve Bromling. Multi-dispatch in the Java virtual machine—design and implementation. In *COOTS 2001 Conference Proceedings*. USENIX Association, January 2001.

[16] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

[17] Rémi Forax, Etienne Duris, and Gilles Roussel. Java multi-method framework. In *TOOLS Pacific '00 Proceedings*. IEEE, November 2000.

[18] Rémi Forax, Etienne Duris, and Gilles Roussel. A simple dispatch technique for pure Java multi-methods. Technical report, Institut d'électronique et d'informatique Gaspard-Monge, 77454 Marne-la-Valle, France, 2001.

[19] Free Software Foundation. GNU C library. `http://www.gnu.org/software/libc/libc.html`. last accessed: February 2002.

[20] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[21] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2000.

[23] Suchitra Gupta, Jeff Hartkopf, and Suresh Ramaswamy. Covariant specialization in Java. *Journal of Object-Oriented Programming*, 13(2), May 2000.

[24] Wade Holst, Duane Szafron, Yuri Leontiev, and Candy Pang. Multi-method dispatch using single-receiver projections. Technical Report 98-03, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1998.

[25] Wade Holst, Duane Szafron, Yuri Leontiev, and Candy Pang. Multi-method dispatch using single-receiver projections. *Software: Practice and Experience*, to appear.

[26] Wade M. Holst. *The Tension between Expressive Power and Method-Dispatch Efficiency*. PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 2000.

[27] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In Norman K. Meyrowitz, editor, *OOPSLA 1986 Conference Proceedings*, volume 21 of *SIGPLAN Notices*, pages 347–349. Association for Computing Machinery, November 1986.

[28] Mark P. Jones. First-class polymorphism with type inference. In *POPL 1997 Symposium Proceedings*, volume 32 of *SIGPLAN Notices*. Association for Computing Machinery, January 1997.

[29] Gary T. Leavens and Todd D. Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA 1998 Conference Proceedings*, volume 33 of *SIGPLAN Notices*, pages 244–258. Association for Computing Machinery, October 1994.

[30] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA 1998 Conference Proceedings*, volume 33 of *SIGPLAN Notices*, pages 36–44. Association for Computing Machinery, October 1998.

[31] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[32] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[33] Steve MacDonald. The MethodThread class for Java. `http://www.cs.ualberta.ca/~stevem/MethodThread2.0/index.html`. last accessed: February 2002.

[34] Thomas J. McCabe and Arthur H. Watson. Software complexity. *Crosstalk, the Journal of Defense Software Engineering*, 7(12):5–9, December 1994.

[35] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT: An open-ended, reflective JIT compile framework for Java. In E. Bertino, editor, *ECOOP 2000 Conference Proceedings*, volume 1850 of *Lecture Notes in Computer Science*, pages 362–387. Springer-Verlag, June 2000.

[36] Chris Rathman. Covariance example in Java. `http://www.angelfire.com/tx4/cus/shapes/covariance.html`. last accessed: February 2002.

[37] Vijay Saraswat. Java is not type-safe. `http://matrix.research.att.com/vj/bug.html`, August 1997. last accessed: February, 2002.

[38] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In Norman K. Meyrowitz, editor, *OOPSLA 1986 Conference Proceedings*, volume 21 of *SIGPLAN Notices*, pages 9–16. Association for Computing Machinery, November 1986.

[39] Doug C. Schmidt and Tim Harrison. Double-checked locking - an object behavioral pattern for initializing and accessing thread-safe objects efficiently, 1996. Pattern Languages of Programming.

[40] Guy L. Steele. *Common Lisp: The Language, 2nd Edition*. Digital Press, Burlington, Massachsetts, 1990.

[41] Christopher Strachey. Fundamental concepts in programming languages. lecture notes for the *International Summer School in Computer Programming*, Copenhagen; republished as [42], August 1967.

[42] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order and Symbolic Computation*, 13(1/2):11–49, April 2000.

[43] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[44] Bjarne Stroustrup. *The C++ Programming Language, 2nd Edition*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[45] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[46] Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.

[47] Sun Microsystems Inc. Adding generics to the Java programming language. `http://java.sun.com/j2se/1.4/compatibility.html` , August 2001. (last accessed: January 2002).

[48] Sun Microsystems Inc. Java(tm) 2 platform compatibility. `http://jcp.org/aboutJava/communityprocess/review/jsr014/` , February 2002. (last accessed: February 2002).

[49] Robert Tolksdorf. Programming languages for the Java virtual machine. `http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html`. last accessed: February 2002.

Appendix A

Performance Raw Results

Here are the individual results for the various double dispatch and multi-dispatch implementations. The results given below summarize 20 repetitions of the tests, giving the average and standard deviation. Each test averaged 1,000,000 dispatches of 49 component-event pairs (each of seven components with each of seven events). Normalization occurs against the UNI implementation.

A.1 Double Dispatch Raw Results

	Event Kernel			Visitor Pattern		
JVM	Time	(σ)	Norm.	Time	(σ)	Norm.
CLASSIC	0.6113	(0.0015)	1.180	0.2352	(0.0001)	0.974
CLIENT	2.0667	(0.0013)	3.990	0.2184	(0.0004)	0.905
SERVER	2.0439	(0.0028)	3.946	0.2247	(0.0003)	0.931
UNI	0.5179	(0.0004)	1.000	0.2413	(0.0001)	1.000
MSA-MI	0.5269	(0.0003)	1.017	0.2365	(0.0004)	0.980
SRP-L-MI	0.5316	(0.0002)	1.026	0.2415	(0.0003)	1.001
SRP-C-MI	0.5324	(0.0003)	1.028	0.2416	(0.0003)	1.001
MSA-INL	0.5419	(0.0004)	1.046	0.2429	(0.0004)	1.006
SRP-L-INL	0.5405	(0.0002)	1.044	0.2455	(0.0003)	1.017
SRP-C-INL	0.5422	(0.0002)	1.046	0.2426	(0.0003)	1.005

Table A.1: Double Dispatch Performance — part I: Event Kernel and Visitor Pattern
(Call-site Dispatch Times in microseconds, smaller is better)

	Typecase			Inlined Typecase		
JVM	Time	(σ)	Norm.	Time	(σ)	Norm.
CLASSIC	0.6965	(0.0005)	1.192	0.5068	(0.0005)	1.259
CLIENT	2.5904	(0.0006)	4.433	2.4243	(0.0008)	6.024
SERVER	2.5778	(0.0022)	4.412	2.4120	(0.0025)	5.994
UNI	0.5843	(0.0005)	1.000	0.4024	(0.0006)	1.000
MSA-MI	0.6029	(0.0003)	1.032	0.4100	(0.0003)	1.019
SRP-L-MI	0.6004	(0.0002)	1.028	0.4090	(0.0003)	1.016
SRP-C-MI	0.6010	(0.0003)	1.029	0.4117	(0.0003)	1.023
MSA-INL	0.6201	(0.0004)	1.061	0.4162	(0.0001)	1.034
SRP-L-INL	0.6174	(0.0004)	1.056	0.4158	(0.0004)	1.033
SRP-C-INL	0.6180	(0.0004)	1.058	0.4181	(0.0004)	1.039

Table A.2: Double Dispatch Performance — part II: Typecases
(Call-site Dispatch Times in microseconds, smaller is better)

	Type field			Inlined Type field		
JVM	Time	(σ)	Norm.	Time	(σ)	Norm.
CLASSIC	0.4258	(0.0004)	0.963	0.2494	(0.0003)	1.028
CLIENT	0.3177	(0.0004)	0.719	0.1731	(0.0001)	0.714
SERVER	0.3462	(0.0002)	0.783	0.1865	(0.0004)	0.769
UNI	0.4420	(0.0005)	1.000	0.2424	(0.0002)	1.000
MSA-MI	0.4340	(0.0001)	0.982	0.2482	(0.0004)	1.024
SRP-L-MI	0.4420	(0.0003)	1.000	0.2450	(0.0004)	1.011
SRP-C-MI	0.4459	(0.0017)	1.008	0.2437	(0.0016)	1.005
MSA-INL	0.4469	(0.0004)	1.011	0.2470	(0.0003)	1.019
SRP-L-INL	0.4448	(0.0003)	1.006	0.2446	(0.0003)	1.009
SRP-C-INL	0.4469	(0.0004)	1.011	0.2410	(0.0001)	0.994

Table A.3: Double Dispatch Performance — part III: Type Fields via Accessor
(Call-site Dispatch Times in microseconds, smaller is better)

	Type field			Inlined Type field		
JVM	Time	(σ)	Norm.	Time	(σ)	Norm.
CLASSIC	0.2962	(0.0002)	0.975	0.0966	(0.0002)	0.960
CLIENT	0.2810	(0.0004)	0.925	0.1160	(0.0004)	1.153
SERVER	0.2931	(0.0003)	0.965	0.1186	(0.0003)	1.178
UNI	0.3037	(0.0004)	1.000	0.1006	(0.0001)	1.000
MSA-MI	0.2965	(0.0003)	0.976	0.1012	(0.0003)	1.006
SRP-L-MI	0.3070	(0.0002)	1.011	0.1013	(0.0003)	1.007
SRP-C-MI	0.3063	(0.0003)	1.008	0.1013	(0.0002)	1.007
MSA-INL	0.3138	(0.0002)	1.033	0.1025	(0.0001)	1.019
SRP-L-INL	0.3130	(0.0004)	1.031	0.1017	(0.0004)	1.011
SRP-C-INL	0.3119	(0.0003)	1.027	0.1017	(0.0004)	1.011

Table A.4: Double Dispatch Performance — part IV: Type Fields via getfield
(Call-site Dispatch Times in microseconds, smaller is better)

A.2 Multi-Dispatch Raw Results

We include data for the SRP algorithm with `MMD_NO_LOCKING`. Although we do not recommend operating the JVM in this mode, it provides a lower bound on the performance of the dispatcher alone.

	Simple MD		Fully MD	
JVM	Time	(σ)	Time	(σ)
MSA-MI	1.4880	(0.0008)	1.4135	(0.0006)
SRP-L-MI	0.5745	(0.0006)	0.3791	(0.0012)
SRP-C-MI	0.4105	(0.0013)	0.2522	(0.0002)
MSA-INL	1.4251	(0.0022)	1.3848	(0.0002)
SRP-L-INL	0.5236	(0.0003)	0.3507	(0.0002)
SRP-C-INL	0.3686	(0.0004)	0.2240	(0.0013)
SRP-NOLOCK-INL	0.3338	(0.0011)	0.2102	(0.0020)

Table A.5: Multi-Dispatch Performance
(Call-site Dispatch Times in microseconds, smaller is better)

Appendix B

Dispatch Evaluation Code

B.1 The Driver Class

```java
import java.util.Date;
class Driver {
    static public String getString() { return "A STRING"; }
    static public void main( String args[] ) {
        Event e[] = { new Event(), new FocusEvent(), new MouseMovedEvent(),
                      new MouseClickedEvent(), new KeyboardEvent(),
                      new InputMethodEvent(), new ComponentEvent() };
        Component c[] = { new Component(), new Button(), new Pane(), new Scroller(),
                          new Container(), new ListBox(), new Chooser() };
        int LOOPS;
        if (args.length < 1) LOOPS = 1000000;
        else LOOPS = Integer.parseInt(args[0]);
        System.out.println("Double dispatch: " + LOOPS + " iterations.");
        Date start, end;
        long overhead, elapsed, dispatch;
        double perdispatch;
        start = new Date();
        for (int i=0; i<LOOPS; i++)
            for (int j=0; j<e.length; j++)
                for (int k=0; k<c.length; k++)
                    getString();
        end = new Date();
        overhead = end.getTime() - start.getTime();
        start = new Date();
        for (int i=0; i<LOOPS; i++)
            for (int j=0; j<e.length; j++)
                for (int k=0; k<c.length; k++)
                    c[k].processEvent(e[j]); //  System.out.println(c[k].processEvent(e[j]));
        end = new Date();
        elapsed = end.getTime() - start.getTime();
        dispatch = elapsed - overhead;
        perdispatch = ((double) (dispatch * 1000)) / ((double) (e.length * c.length * LOOPS));
        System.out.println(" elapsed: " + elapsed + " ms, overhead: " + overhead
            + " ms, dispatch: " + dispatch + " ms, perdispatch: " + perdispatch + "us.");
    }
}
```

Figure B.1: Double Dispatch Driver

B.2 Uni Dispatch

```
class Event { ... }
class FocusEvent        extends Event { ... }
class MouseMovedEvent   extends Event { ... }
class MouseClickedEvent extends Event { ... }
class KeyboardEvent     extends Event { ... }
class InputMethodEvent  extends Event { ... }
class ComponentEvent    extends Event { ... }
class Component {
    public void processEvent(Event e) { getString(); }
}
class Button    extends Component { ... }
class Pane      extends Component { ... }
class Scroller  extends Component { ... }
class Container extends Component { ... }
class ListBox   extends Component { ... }
class Chooser   extends Component { ... }
```

Figure B.2: Uni-Dispatch Basic Structure

B.3 Double Dispatch

B.3.1 Visitor Pattern

```
class Event {
    String processEvent(Component c) { return "Event x Component"; }
    String processEvent(Button b)    { return "Event x Button";    }
    String processEvent(Pane p)      { return "Event x Pane";      }
    String processEvent(Scroller s)  { return "Event x Scroller";  }
    String processEvent(Container c) { return "Event x Container"; }
    String processEvent(ListBox lb)  { return "Event x ListBox";   }
    String processEvent(Chooser c)   { return "Event x Chooser";   }
}
class FocusEvent extends Event {
    String processEvent(Component c) { return "FocusEvent x Component"; }
    String processEvent(Button b)    { return "FocusEvent x Button";    }
    String processEvent(Pane p)      { return "FocusEvent x Pane";      }
    String processEvent(Scroller s)  { return "FocusEvent x Scroller";  }
    String processEvent(Container c) { return "FocusEvent x Container"; }
    String processEvent(ListBox lb)  { return "FocusEvent x ListBox";   }
    String processEvent(Chooser c)   { return "FocusEvent x Chooser";   }
}
//  ...remaining event subclasses omitted ...
class Component {
    String processEvent( Event e)              { return  e.processEvent(this); }
    String processEvent( FocusEvent f)         { return  f.processEvent(this); }
    String processEvent( MouseMovedEvent mm)   { return mm.processEvent(this); }
    String processEvent( MouseClickedEvent mc) { return mc.processEvent(this); }
    String processEvent( KeyboardEvent k)      { return  k.processEvent(this); }
    String processEvent( InputMethodEvent im)  { return im.processEvent(this); }
    String processEvent( ComponentEvent c)     { return  c.processEvent(this); }
}
class Button extends Component {
    String processEvent( Event e)              { return  e.processEvent(this); }
    String processEvent( FocusEvent f)         { return  f.processEvent(this); }
    String processEvent( MouseMovedEvent mm)   { return mm.processEvent(this); }
    String processEvent( MouseClickedEvent mc) { return mc.processEvent(this); }
    String processEvent( KeyboardEvent k)      { return  k.processEvent(this); }
    String processEvent( InputMethodEvent im)  { return im.processEvent(this); }
    String processEvent( ComponentEvent c)     { return  c.processEvent(this); }
}
//  ...remaining component subclasses omitted ...
```

Figure B.3: Visitor Pattern

B.3.2 Typecases

```
class Event { ... }
class FocusEvent extends Event { ... }
class MouseMovedEvent extends Event { ... }
class MouseClickedEvent extends Event { ... }
class KeyboardEvent extends Event { ... }
class InputMethodEvent extends Event { ... }
class ComponentEvent extends Event { ... }
class Component {
    String processEvent( Event e) {
        if (e instanceof FocusEvent)             return this.processEvent((FocusEvent) e);             }
        else if (e instanceof MouseMovedEvent)   return this.processEvent((MouseMovedEvent) e);        }
        else if (e instanceof MouseClickedEvent) return this.processEvent((MouseClickedEvent) e);      }
        else if (e instanceof KeyboardEvent)     return this.processEvent((KeyboardEvent) e);          }
        else if (e instanceof InputMethodEvent)  return this.processEvent((InputMethodEvent) e);       }
        else if (e instanceof ComponentEvent)    return this.processEvent((ComponentEvent) e);         }
        else /* (e instanceof Event) */          return this._processEvent(e);                         }
    }
    String _processEvent( Event e)              { return              "Event x Component"; }
    String processEvent( FocusEvent f)          { return         "FocusEvent x Component"; }
    String processEvent( MouseMovedEvent mm)    { return    "MouseMovedEvent x Component"; }
    String processEvent( MouseClickedEvent mc)  { return  "MouseClickedEvent x Component"; }
    String processEvent( KeyboardEvent k)       { return      "KeyboardEvent x Component"; }
    String processEvent( InputMethodEvent im)   { return   "InputMethodEvent x Component"; }
    String processEvent( ComponentEvent c)      { return     "ComponentEvent x Component"; }
}
class Button extends Component {
    String _processEvent( Event e)              { return              "Event x Button"; }
    String processEvent( FocusEvent f)          { return         "FocusEvent x Button"; }
    String processEvent( MouseMovedEvent mm)    { return    "MouseMovedEvent x Button"; }
    String processEvent( MouseClickedEvent mc)  { return  "MouseClickedEvent x Button"; }
    String processEvent( KeyboardEvent k)       { return      "KeyboardEvent x Button"; }
    String processEvent( InputMethodEvent im)   { return   "InputMethodEvent x Button"; }
    String processEvent( ComponentEvent c)      { return     "ComponentEvent x Button"; }
}
//  ...remaining component subclasses omitted ...
```

Figure B.4: Typecases

B.3.3 Inlined Typecases

```
class Event { }
class FocusEvent extends Event { }
class MouseMovedEvent extends Event { }
class MouseClickedEvent extends Event { }
class KeyboardEvent extends Event { }
class InputMethodEvent extends Event { }
class ComponentEvent extends Event { }
class Component {
    String processEvent( Event e) {
        if (e instanceof FocusEvent)             { return        "FocusEvent x Component"; }
        else if (e instanceof MouseMovedEvent)   { return   "MouseMovedEvent x Component"; }
        else if (e instanceof MouseClickedEvent) { return "MouseClickedEvent x Component"; }
        else if (e instanceof KeyboardEvent)     { return     "KeyboardEvent x Component"; }
        else if (e instanceof InputMethodEvent)  { return  "InputMethodEvent x Component"; }
        else if (e instanceof ComponentEvent)    { return    "ComponentEvent x Component"; }
        else /* (e instanceof Event) */          { return             "Event x Component"; }
    }
}
class Button extends Component {
    String processEvent( Event e) {
        if (e instanceof FocusEvent)             { return        "FocusEvent x Button"; }
        else if (e instanceof MouseMovedEvent)   { return   "MouseMovedEvent x Button"; }
        else if (e instanceof MouseClickedEvent) { return "MouseClickedEvent x Button"; }
        else if (e instanceof KeyboardEvent)     { return     "KeyboardEvent x Button"; }
        else if (e instanceof InputMethodEvent)  { return  "InputMethodEvent x Button"; }
        else if (e instanceof ComponentEvent)    { return    "ComponentEvent x Button"; }
        else /* (e instanceof Event) */          { return             "Event x Button"; }
    }
}
//  ...remaining component subclasses omitted ...
```

Figure B.5: Inlined Typecases

B.3.4 Typefields via Accessors

```
class Event {
    static final int EVENT     = 0;
    static final int FOCUS     = 1;
    static final int MOVED     = 2;
    static final int CLICK     = 3;
    static final int KEYBOARD  = 4;
    static final int INPUT     = 5;
    static final int COMPONENT = 6;
    final int id;

    final int getID() { return this.id;  }
    Event(int i)      { this.id = i;     }
    Event()           { this.id = EVENT; }
}
class FocusEvent extends Event {
    FocusEvent() { super(Event.FOCUS); }
}
//  ...remaining event subclasses omitted ...
class Component {
    String processEvent( Event e) {
        switch (e.getID()) {
        case Event.EVENT:     return this._processEvent(e);
        case Event.FOCUS:     return this.processEvent((FocusEvent) e);
        case Event.MOVED:     return this.processEvent((MouseMovedEvent) e);
        case Event.CLICK:     return this.processEvent((MouseClickedEvent) e);
        case Event.KEYBOARD:  return this.processEvent((KeyboardEvent) e);
        case Event.INPUT:     return this.processEvent((InputMethodEvent) e);
        case Event.COMPONENT: return this.processEvent((ComponentEvent) e);
        }
        return "";
    }
    String _processEvent( Event e)             { return             "Event x Component"; }
    String processEvent( FocusEvent f)         { return        "FocusEvent x Component"; }
    String processEvent( MouseMovedEvent mm)   { return   "MouseMovedEvent x Component"; }
    String processEvent( MouseClickedEvent mc) { return "MouseClickedEvent x Component"; }
    String processEvent( KeyboardEvent k)      { return     "KeyboardEvent x Component"; }
    String processEvent( InputMethodEvent im)  { return  "InputMethodEvent x Component"; }
    String processEvent( ComponentEvent c)     { return    "ComponentEvent x Component"; }
}
class Button extends Component {
    String _processEvent( Event e)             { return             "Event x Button"; }
    String processEvent( FocusEvent f)         { return        "FocusEvent x Button"; }
    String processEvent( MouseMovedEvent mm)   { return   "MouseMovedEvent x Button"; }
    String processEvent( MouseClickedEvent mc) { return "MouseClickedEvent x Button"; }
    String processEvent( KeyboardEvent k)      { return     "KeyboardEvent x Button"; }
    String processEvent( InputMethodEvent im)  { return  "InputMethodEvent x Button"; }
    String processEvent( ComponentEvent c)     { return    "ComponentEvent x Button"; }
}
//  ...remaining component subclasses omitted ...
```

Figure B.6: Typefields via Accessors

B.3.5 Inlined Typefields via Accessors

```
class Event {
    static final int EVENT     = 0;
    static final int FOCUS     = 1;
    static final int MOVED     = 2;
    static final int CLICK     = 3;
    static final int KEYBOARD  = 4;
    static final int INPUT     = 5;
    static final int COMPONENT = 6;
    final int id;

    final int getID() { return this.id; }
    Event(int i)      { this.id = i; }
    Event()           { this.id = EVENT; }
}
class FocusEvent extends Event {
    FocusEvent() { super(Event.FOCUS); }
}
//  ...remaining event subclasses omitted ...
class Component {
    String processEvent( Event e) {
        switch (e.getID()) {
        case Event.EVENT:     return             "Event x Component";
        case Event.FOCUS:     return        "FocusEvent x Component";
        case Event.MOVED:     return   "MouseMovedEvent x Component";
        case Event.CLICK:     return "MouseClickedEvent x Component";
        case Event.KEYBOARD:  return     "KeyboardEvent x Component";
        case Event.INPUT:     return  "InputMethodEvent x Component";
        case Event.COMPONENT: return    "ComponentEvent x Component";
        }
        return "";
    }
}
class Button extends Component {
    String processEvent( Event e) {
        switch (e.getID()) {
        case Event.EVENT:     return             "Event x Button";
        case Event.FOCUS:     return        "FocusEvent x Button";
        case Event.MOVED:     return   "MouseMovedEvent x Button";
        case Event.CLICK:     return "MouseClickedEvent x Button";
        case Event.KEYBOARD:  return     "KeyboardEvent x Button";
        case Event.INPUT:     return  "InputMethodEvent x Button";
        case Event.COMPONENT: return    "ComponentEvent x Button";
        }
        return "";
    }
//  ...remaining component subclasses omitted ...
```

Figure B.7: Inlined Direct Typefields

B.3.6 Direct Typefields

```
class Event {
    static final int EVENT     = 0;
    static final int FOCUS     = 1;
    static final int MOVED     = 2;
    static final int CLICK     = 3;
    static final int KEYBOARD  = 4;
    static final int INPUT     = 5;
    static final int COMPONENT = 6;
    final int id;

    Event(int i)        { this.id = i; }
    Event()             { this.id = EVENT; }
}
class FocusEvent extends Event {
    FocusEvent() { super(Event.FOCUS); }
}
//  ...remaining event subclasses omitted ...
class Component {
class Component {
    String processEvent( Event e) {
        switch (e.id) {
        case Event.EVENT:     return this._processEvent(e);
        case Event.FOCUS:     return this.processEvent((FocusEvent) e);
        case Event.MOVED:     return this.processEvent((MouseMovedEvent) e);
        case Event.CLICK:     return this.processEvent((MouseClickedEvent) e);
        case Event.KEYBOARD:  return this.processEvent((KeyboardEvent) e);
        case Event.INPUT:     return this.processEvent((InputMethodEvent) e);
        case Event.COMPONENT: return this.processEvent((ComponentEvent) e);
        }
        return "";
    }
    String _processEvent( Event e)            { return             "Event x Component"; }
    String processEvent( FocusEvent f)        { return        "FocusEvent x Component"; }
    String processEvent( MouseMovedEvent mm)  { return   "MouseMovedEvent x Component"; }
    String processEvent( MouseClickedEvent mc) { return "MouseClickedEvent x Component"; }
    String processEvent( KeyboardEvent k)     { return     "KeyboardEvent x Component"; }
    String processEvent( InputMethodEvent im) { return  "InputMethodEvent x Component"; }
    String processEvent( ComponentEvent c)    { return    "ComponentEvent x Component"; }
}
class Button extends Component {
    String _processEvent( Event e)            { return             "Event x Button"; }
    String processEvent( FocusEvent f)        { return        "FocusEvent x Button"; }
    String processEvent( MouseMovedEvent mm)  { return   "MouseMovedEvent x Button"; }
    String processEvent( MouseClickedEvent mc) { return "MouseClickedEvent x Button"; }
    String processEvent( KeyboardEvent k)     { return     "KeyboardEvent x Button"; }
    String processEvent( InputMethodEvent im) { return  "InputMethodEvent x Button"; }
    String processEvent( ComponentEvent c)    { return    "ComponentEvent x Button"; }
}
//  ...remaining component subclasses omitted ...
```

Figure B.8: Direct Typefields

B.3.7 Inlined Direct Typefields

```
class Event {
    static final int EVENT     = 0;
    static final int FOCUS     = 1;
    static final int MOVED     = 2;
    static final int CLICK     = 3;
    static final int KEYBOARD  = 4;
    static final int INPUT     = 5;
    static final int COMPONENT = 6;
    final int id;

    Event(int i)        { this.id = i; }
    Event()             { this.id = EVENT; }
}
class FocusEvent extends Event {
    FocusEvent() { super(Event.FOCUS); }
}
//  ...remaining event subclasses omitted ...
class Component {
    String processEvent( Event e) {
        switch (e.id) {
        case Event.EVENT:     return             "Event x Component";
        case Event.FOCUS:     return        "FocusEvent x Component";
        case Event.MOVED:     return   "MouseMovedEvent x Component";
        case Event.CLICK:     return "MouseClickedEvent x Component";
        case Event.KEYBOARD:  return     "KeyboardEvent x Component";
        case Event.INPUT:     return  "InputMethodEvent x Component";
        case Event.COMPONENT: return    "ComponentEvent x Component";
        }
        return "";
    }
}
class Button extends Component {
    String processEvent( Event e) {
        switch (e.getID()) {
        case Event.EVENT:     return             "Event x Button";
        case Event.FOCUS:     return        "FocusEvent x Button";
        case Event.MOVED:     return   "MouseMovedEvent x Button";
        case Event.CLICK:     return "MouseClickedEvent x Button";
        case Event.KEYBOARD:  return     "KeyboardEvent x Button";
        case Event.INPUT:     return  "InputMethodEvent x Button";
        case Event.COMPONENT: return    "ComponentEvent x Button";
        }
        return "";
    }
//  ...remaining component subclasses omitted ...
```

Figure B.9: Inlined Direct Typefields

B.3.8 Event Dispatch Kernel

```
class Event { ... }
class FocusEvent        extends Event { ... }
class MouseMovedEvent   extends Event { ... }
class MouseClickedEvent extends Event { ... }
class KeyboardEvent     extends Event { ... }
class InputMethodEvent  extends Event { ... }
class ComponentEvent    extends Event { ... }
class Component {
    public void processEvent(Event e) {
        if (e instanceof FocusEvent) processFocusEvent((FocusEvent)e);
        else if (e instanceof MouseEvent)
            switch(e.getID()) {
                case MouseEvent.MOUSE_PRESSED: processMouseEvent((MouseClickedEvent)e); break;
                case MouseEvent.MOUSE_MOVED:   processMouseMovedEvent((MouseMovedEvent)e); break;
            }
        else if (e instanceof KeyEvent) processKeyEvent((KeyEvent)e);
        else if (e instanceof ComponentEvent) processComponentEvent((ComponentEvent)e);
        else if (e instanceof InputMethodEvent) processInputMethodEvent((InputMethodEvent)e);
        else _processEvent(e);
    }
    String _processEvent( Event e)                          { return             "Event x Component"; }
    String processFocusEvent( FocusEvent f)                 { return        "FocusEvent x Component"; }
    String processMouseMovedEvent( MouseMovedEvent mm)      { return   "MouseMovedEvent x Component"; }
    String processMouseClickedEvent( MouseClickedEvent mc)  { return "MouseClickedEvent x Component"; }
    String processKeyEvent( KeyboardEvent k)                { return     "KeyboardEvent x Component"; }
    String processInputMethodEvent( InputMethodEvent im)    { return  "InputMethodEvent x Component"; }
    String processComponentEvent( ComponentEvent c)         { return    "ComponentEvent x Component"; }
}
class Button extends Component {
    String _processEvent( Event e)                          { return             "Event x Button"; }
    String processFocusEvent( FocusEvent f)                 { return        "FocusEvent x Button"; }
    String processMouseMovedEvent( MouseMovedEvent mm)      { return   "MouseMovedEvent x Button"; }
    String processMouseClickedEvent( MouseClickedEvent mc)  { return "MouseClickedEvent x Button"; }
    String processKeyEvent( KeyboardEvent k)                { return     "KeyboardEvent x Button"; }
    String processInputMethodEvent( InputMethodEvent im)    { return  "InputMethodEvent x Button"; }
    String processComponentEvent( ComponentEvent c)         { return    "ComponentEvent x Button"; }
}
//  ...remaining component subclasses omitted ...
```

Figure B.10: Event Dispatch Kernel

B.4 Multi-Dispatch

B.4.1 Simple Multi-Dispatch

```
class Event { ... }
class FocusEvent        extends Event { ... }
class MouseMovedEvent   extends Event { ... }
class MouseClickedEvent extends Event { ... }
class KeyboardEvent     extends Event { ... }
class InputMethodEvent  extends Event { ... }
class ComponentEvent    extends Event { ... }
class Component {
    public void processEvent(MouseEvent e) {
        switch(e.getID()) {
            case MouseEvent.MOUSE_PRESSED: processMouseClickedEvent(e); break;
            case MouseEvent.MOUSE_MOVED:   processMouseMovedEvent(e); break;
        }
    }
    String processEvent( Event e)                               { return             "Event x Component"; }
    String processEvent( FocusEvent f)                          { return        "FocusEvent x Component"; }
    String processMouseMovedEvent( MouseMovedEvent mm)          { return   "MouseMovedEvent x Component"; }
    String processMouseClickedEvent( MouseClickedEvent mc)      { return "MouseClickedEvent x Component"; }
    String processEvent( KeyboardEvent k)                       { return     "KeyboardEvent x Component"; }
    String processEvent( InputMethodEvent im)                   { return  "InputMethodEvent x Component"; }
    String processEvent( ComponentEvent c)                      { return    "ComponentEvent x Component"; }
}
class Button extends Component {
    String processEvent( Event e)                               { return             "Event x Button"; }
    String processEvent( FocusEvent f)                          { return        "FocusEvent x Button"; }
    String processMouseMovedEvent( MouseMovedEvent mm)          { return   "MouseMovedEvent x Button"; }
    String processMouseClickedEvent( MouseClickedEvent mc)      { return "MouseClickedEvent x Button"; }
    String processEvent( KeyboardEvent k)                       { return     "KeyboardEvent x Button"; }
    String processEvent( InputMethodEvent im)                   { return  "InputMethodEvent x Button"; }
    String processEvent( ComponentEvent c)                      { return    "ComponentEvent x Button"; }
}
//  ...remaining component subclasses omitted ...
```

Figure B.11: Simple Multi-Dispatch

B.4.2 Fully Multi-Dispatch

```
class Event { ... }
class FocusEvent         extends Event { ... }
class MouseMovedEvent    extends Event { ... }
class MouseClickedEvent  extends Event { ... }
class KeyboardEvent      extends Event { ... }
class InputMethodEvent   extends Event { ... }
class ComponentEvent     extends Event { ... }
class Component {
    String processEvent( Event e)              { return             "Event x Component"; }
    String processEvent( FocusEvent f)         { return        "FocusEvent x Component"; }
    String processEvent( MouseMovedEvent mm)   { return   "MouseMovedEvent x Component"; }
    String processEvent( MouseClickedEvent mc) { return "MouseClickedEvent x Component"; }
    String processEvent( KeyboardEvent k)      { return     "KeyboardEvent x Component"; }
    String processEvent( InputMethodEvent im)  { return  "InputMethodEvent x Component"; }
    String processEvent( ComponentEvent c)     { return    "ComponentEvent x Component"; }
}
class Button extends Component {
    String processEvent( Event e)              { return             "Event x Button"; }
    String processEvent( FocusEvent f)         { return        "FocusEvent x Button"; }
    String processEvent( MouseMovedEvent mm)   { return   "MouseMovedEvent x Button"; }
    String processEvent( MouseClickedEvent mc) { return "MouseClickedEvent x Button"; }
    String processEvent( KeyboardEvent k)      { return     "KeyboardEvent x Button"; }
    String processEvent( InputMethodEvent im)  { return  "InputMethodEvent x Button"; }
    String processEvent( ComponentEvent c)     { return    "ComponentEvent x Button"; }
}
//  ...remaining component subclasses omitted ...
```

Figure B.12: Fully Multi-Dispatch

Appendix C

Test Suite

C.1 ByteCode Tests

C.1.1 `IVQ`

```
class A extends Object {}
class B extends A {}
class SuperIVQ implements VirtualMultiDispatchable {
  String mmd(A a) { return "SuperIVQ::mmd(A)"; }
}
class IVQ extends SuperIVQ {
  String mmd(B b) { return "IVQ::mmd(B)"; }
  public static void main(String[] args) {
    boolean print = true;
    int LOOPS = 1;
    for (int i=0; i<args.length; i++)
      if (args[i].equals("-p")) print = !print;
      else
        try {
          LOOPS = Integer.parseInt(args[i]);
          if (LOOPS > 5) print = false;
        } catch (NumberFormatException nfe) {
          System.out.println("Not a number:  " + args[i]);
        }
    SuperIVQ i = new IVQ();
    A[] a = new A[2];
    a[0] = new A(); System.out.println("Made " + a[0]);
    a[1] = new B(); System.out.println("Made " + a[1]);
    for (int j=0; j<LOOPS; j++)
      for (int k=0; k<2; k++)
        if (print) System.out.println(i.mmd(a[k]));
        else i.mmd(a[k]);
  }
}
          //  --->
          // Made A@c7ef55fd
          // Made B@c7ef8466
          // SuperIVQ::mmd(A)
          // IVQ::mmd(B)
```

Figure C.1: `IVQ` Test

C.1.2 IVQW

```
class A extends Object {}
class B extends A {}
class SuperIVQW implements VirtualMultiDispatchable {
  // need enough methods to force a wide bytecode
  String AA() { return "AA"; } String AB() { return "AB"; }
  String AC() { return "AC"; } String AD() { return "AD"; }
  String AE() { return "AE"; } String AF() { return "AF"; }
  // ... 248 methods omitted ...
  String JU() { return "JU"; } String JV() { return "JV"; }
  String JW() { return "JW"; } String JX() { return "JX"; }
  String JY() { return "JY"; } String JZ() { return "JZ"; }
  String mmd(A a) { return "SuperIVQW::mmd(A)"; }
}
class IVQW extends SuperIVQW {
  String mmd(B b) { return "IVQW::mmd(B)"; }
  public static void main(String[] args) {
    boolean print = true;
    int LOOPS = 1;
    for (int i=0; i<args.length; i++)
      if (args[i].equals("-p")) print = !print;
      else
        try {
          LOOPS = Integer.parseInt(args[i]);
          if (LOOPS > 5) print = false;
        } catch (NumberFormatException nfe) {
          System.out.println("Not a number:  " + args[i]);
        }
    SuperIVQW i = new IVQW();
    A[] a = new A[2];
    a[0] = new A(); System.out.println("Made " + a[0]);
    a[1] = new B(); System.out.println("Made " + a[1]);
    for (int j=0; j<LOOPS; j++)
      for (int k=0; k<2; k++)
        if (print) System.out.println(i.mmd(a[k]));
        else i.mmd(a[k]);
  }
}
          //  --->
          // Made A@c7ef55f4
          // Made B@c7ef8466
          // SuperIVQW::mmd(A)
          // IVQW::mmd(B)
```

Figure C.2: `IVQW` Test

C.1.3 IVOW

```
class A extends Object {}
class B extends A {}
class SuperIVOW implements VirtualMultiDispatchable {
  boolean equals(A a) {
    System.out.println("SuperIVOW::equal(A)");
    return super.equals(a);
  }
}
class IVOW extends SuperIVOW {
  boolean equals(B b) {
    System.out.println("IVOW::equal(B)");
    return super.equals(b);
  }
  public static void main(String[] args) {
    boolean print = true;
    int LOOPS = 1;
    for (int i=0; i<args.length; i++)
      if (args[i].equals("-p")) print = !print;
      else
        try {
          LOOPS = Integer.parseInt(args[i]);
          if (LOOPS > 5) print = false;
        } catch (NumberFormatException nfe) {
          System.out.println("Not a number:  " + args[i]);
        }
    Object[] i = new Object[3];
    Object[] o = new Object[3];
    i[0] = new Object();
    i[1] = new SuperIVOW();
    i[2] = new IVOW();
    o[0] = new Object();
    o[1] = new A();
    o[2] = new B();
    for (int j=0; j<LOOPS; j++)
      for (int k=0; k<3; k++)
        for (int l=0; l<3; l++)
          if (print) System.out.println(i[k].equals(o[l]));
          else i[k].equals(o[l]);
  }
}
          //  --->
          // false
          // false
          // false
          // false
          // SuperIVOW::equal(A)
          // false
          // SuperIVOW::equal(A)
          // false
          // false
          // SuperIVOW::equal(A)
          // false
          // IVOW::equal(B)
          // SuperIVOW::equal(A)
          // false
```

Figure C.3: `IVOW` Test

C.1.4 ISQ

```
class A extends Object {}
class B extends A {}
class ISQ implements StaticMultiDispatchable {
  static String mmd(A a) { return "ISQ::mmd(A)"; }
  static String mmd(B b) { return "ISQ::mmd(B)"; }
  public static void main(String[] args) {
    boolean print = true;
    int LOOPS = 1;
    for (int i=0; i<args.length; i++)
      if (args[i].equals("-p")) print = !print;
      else
        try {
          LOOPS = Integer.parseInt(args[i]);
          if (LOOPS > 5) print = false;
        } catch (NumberFormatException nfe) {
          System.out.println("Not a number:  " + args[i]);
        }
    A[] a = new A[2];
    a[0] = new A();
    a[1] = new B();
    for (int j=0; j<LOOPS; j++)
      for (int k=0; k<2; k++)
        if (print) System.out.println(mmd(a[k]));
        else mmd(a[k]);
  }
}
        //  --->
        // ISQ::mmd(A)
        // ISQ::mmd(B)
```

Figure C.4: ISQ Test

C.1.5 INVQ

```
class A extends Object {}
class B extends A {}
class SuperINVQ implements SpecialMultiDispatchable {
  SuperINVQ(A a) { System.out.println("SuperINVQ::<init>(A)"); }
  SuperINVQ(B b) { System.out.println("SuperINVQ::<init>(B)"); }
}
class INVQ extends SuperINVQ {
  private final String mmd(A a) { return "INVQ::mmd(A)"; }
  private final String mmd(B b) { return "INVQ::mmd(B)"; }
  INVQ(A a) {
    super(a);
    System.out.println("INVQ::<init>(A)");
  }
  public static void main(String[] args) {
    boolean print = true;
    int LOOPS = 1;
    for (int i=0; i<args.length; i++)
      if (args[i].equals("-p")) print = !print;
      else
        try {
          LOOPS = Integer.parseInt(args[i]);
          if (LOOPS > 5) print = false;
        } catch (NumberFormatException nfe) {
          System.out.println("Not a number:  " + args[i]);
        }
    INVQ i = new INVQ(new A());
    new INVQ(new B());
    A[] a = new A[2];
    a[0] = new A(); System.out.println("Made " + a[0]);
    a[1] = new B(); System.out.println("Made " + a[1]);
    for (int j=0; j<LOOPS; j++)
      for (int k=0; k<2; k++)
        if (print) System.out.println(i.mmd(a[k]));
        else i.mmd(a[k]);
  }
}
          //  --->
          // SuperINVQ::<init>(A)
          // INVQ::<init>(A)
          // SuperINVQ::<init>(B)
          // INVQ::<init>(A)
          // Made A@489aaced
          // Made B@489a742d
          // INVQ::mmd(A)
          // INVQ::mmd(B)
```

Figure C.5: `INVQ` Test

C.1.6 INVSQ

```
class A extends Object {}
class B extends A {}
class supersuperINVSQ implements SpecialMultiDispatchable {
  String mmd(A a) { return "supersuperINVSQ::mmd(A)"; }
  String mmd(B b) { return "supersuperINVSQ::mmd(B)"; }
}
class superINVSQ extends supersuperINVSQ {
  String mmd(B b) { return "superINVSQ::mmd(B)"; }
}
class INVSQ extends superINVSQ {
  String mmd(A a) { return super.mmd(a); }
  String mmd(B b) { return super.mmd(b); }
  public static void main(String[] args) {
    boolean print = true;
    int LOOPS = 1;
    for (int i=0; i<args.length; i++)
      if (args[i].equals("-p")) print = !print;
      else
        try {
          LOOPS = Integer.parseInt(args[i]);
          if (LOOPS > 5) print = false;
        } catch (NumberFormatException nfe) {
          System.out.println("Not a number:  " + args[i]);
        }
      }
    }
    INVSQ i = new INVSQ();
    A[] a = new A[2];
    a[0] = new A(); System.out.println("Made " + a[0]);
    a[1] = new B(); System.out.println("Made " + a[1]);
    for (int j=0; j<LOOPS; j++)
      for (int k=0; k<2; k++)
        if (print) System.out.println(i.mmd(a[k]));
        else i.mmd(a[k]);
  }
}
          //  --->
          // Made A@489aaeb5
          // Made B@489a750c
          // supersuperINVSQ::mmd(A)
          // superINVSQ::mmd(B)
```

Figure C.6: `INVSQ` Test

C.2 Integrated Tests

C.2.1 Multi-Dispatch Diamond

```
class D1 implements VirtualMultiDispatchable {
  void m(D1 x) { System.out.println("D1.m(D1)"); }
  void m(D2 y) { System.out.println("D1.m(D2)"); }
}
class D2 extends D1 {
  void m(D1 x) { System.out.println("D2.m(D1)"); }
  void m(D2 y) { System.out.println("D2.m(D2)"); }
}
class Diamond {
  static public void main(String args[]) {
    D1 x = new D1();
    D1 y = new D2();
    x.m(x);
    x.m(y);
    y.m(x);
    y.m(y);
    y.m(null);
  }
}
          //  --->
          // D1.m(D1)
          // D1.m(D2)
          // D2.m(D1)
          // D2.m(D2)
          // D2.m(D2)
```

Figure C.7: Multi-Dispatch Diamond

C.2.2 Array Types

```
class A {}
class B extends A implements Cloneable {}
class C {}
interface I {}
class D implements I {}
public class ArrTest implements VirtualMultiDispatchable {
  static final int NUM_TESTS = 11;
  public static void main( String[] args )
  {
    ArrTest test = new ArrTest();
    for ( int i = 0; i < NUM_TESTS; i++ )
      try {
        System.out.println("test.m(" + test.getParameter(i) + ")");
        test.m( test.getParameter( i ) );
      } catch ( AmbiguousMethodError ame ) {
        System.out.println("AME raised:");
        for (int j=0; j<ame.methods.length; j++)
          System.out.println("" + ame.methods[j]);
      } catch ( Exception e ) {
        System.out.println("Caught exception " + e + " at test " + i + ".");
      }
  }
  public Object getParameter( int i )
  {
    switch ( i ) {
    case 0:  return new Object();
    case 1:  return new Object[1];
    case 2:  return new A();
    case 3:  return new A[1];
    case 4:  return new B();
    case 5:  return new B[1];
    case 6:  return new C();
    case 7:  return new C[1];
    case 8:  return new Integer(1);
    case 9:  return new D[1];
    default:  return null;
    }
  }
  public void m( Object x )   { System.out.println( "Object reached:  "    + x ); }
  public void m( Object[] x ) { System.out.println( "Object[] reached:  "  + x ); }
  public void m( Integer x )  { System.out.println( "Integer reached:  "   + x ); }
  public void m( A[] x )      { System.out.println( "A[] reached:  "       + x ); }
  public void m(Cloneable x)  { System.out.println( "Cloneable reached:  " + x ); }
  public void m(I[] x)        { System.out.println( "I[] reached:  "       + x ); }
}
        //  --->
        // test.m(java.lang.Object@b910e084)
        // Object reached:  java.lang.Object@b9104e44
        // test.m([Ljava.lang.Object;@b910df1c)
        // Object[] reached:  [Ljava.lang.Object;@b910f1c1
        // test.m(A@b910df7f)
        // Object reached:  A@b910dff4
        // test.m([LA;@b910f299)
        // A[] reached:  [LA;@b910f30e
        // test.m(B@b910f64a)
        // Cloneable reached:  B@b910efdb
        // test.m([LB;@b910f06b)
        // A[] reached:  [LB;@b910f437
        // test.m(C@b910f41c)
        // Object reached:  C@b910ea5b
        // test.m([LC;@b910f46d)
        // Object[] reached:  [LC;@b910f4c7
        // test.m(1)
        // Integer reached 1
        // test.m([LD;@b910ee9f)
        // I[] reached:  [LD;@b910ef4a
        // test.m(null)
        // AME raised:
        //      public void ArrTest.m(I[])
        //      public void ArrTest.m(java.lang.Integer)
        //      public void ArrTest.m(A[])
```

Figure C.8: Array Types Test

C.2.3 Invoke-virtual Semantics

```
// VirtualTest.java -- test that virtual multi-dispatch works
// no methods - purely used for argument hierarchy
class A {}
class B extends A {}
class C extends B {}
class U {
    String method()            { return "U::method()";     }
    String method(A a)         { return "U::method(A)";    }
    String method(B b[])       { return "U::method([B)";   }
    String method(A a1, A a2)  { return "U::method(A,A)";  }
    String method(B b, A a)    { return "U::method(B,A)";  }
 }
class V extends U implements VirtualMultiDispatchable {
    String method()            { return "V::method()";     }
    String method(A a[])       { return "A::method([A)";   }
    String method(B b)         { return "V::method(B)";    }
    String method(A a1, A a2)  { return "V::method(A,A)";  }
    String method(A a, B b)    { return "V::method(A,B)";  }
 }
class W extends V {
    String method()            { return "W::method()";     }
    Integer method(C c)        { return new Integer(0);    }
    String method(B b, C c)    { return "W::method(B,C)";  }
 }
```

Figure C.9: `invoke-virtual` Semantics Test (Part I)

```
class VirtualTest {
    A args[];
    String result;
    boolean ambiguousmethod;
    boolean illegalreturntype;
    VirtualTest(String a, String r, boolean ame, boolean irt) {
        args = new A[a.length()];
        for (int i=0; i<a.length(); i++)
            switch (a.charAt(i)) {
            case 'A': args[i] = new A(); break;
            case 'B': args[i] = new B(); break;
            case 'C': args[i] = new C(); break;
            case 'n': args[i] = null;    break;
            }
        result = r;
        ambiguousmethod = ame;
        illegalreturntype = irt;
    }
    boolean test(U u, boolean print) {
        boolean sa = false;
        boolean si = false;
        String r = "***INVALID***";
        try {
            switch (args.length) {
            case 0: r = u.method();                    break;
            case 1: r = u.method(args[0]);             break;
            case 2: r = u.method(args[0], args[1]); break;
            }
            if (print) System.out.println("=====" + r);
        } catch (AmbiguousMethodError a) {
            if (print) {
                System.out.println("AME seen with " + a.methods.length + " methods applying.");
                for (int i=0; i<a.methods.length; i++)
                System.out.println("" + a.methods[i]);
            }
            if (false == ambiguousmethod) {
                System.err.println("Error: unexpected AmbiguousMethod thrown");
                return false;
            } else sa = true;
        } catch (IllegalReturnTypeError i) {
            if (false == illegalreturntype) {
                System.err.println("Error: unexpected IllegalReturnType thrown");
                return false;
            } else si = true;
        } catch (Exception e) {
               System.err.println("Error: unexpected " + e + " thrown");
               return false;
        } finally {
            if (sa != ambiguousmethod) {
                System.err.println("Error: expected AmbiguousMethod not thrown");
                return false;
            }
            if (si != illegalreturntype) {
                System.err.println("Error: expected IllegalReturnType not thrown");
                return false;
            }
            if (!sa && !si && r != result) {
                System.err.println("Error: |" + r + "| returned, expected |" + result + "|");
                return false;
            }
        }
        return true;
    }
}
```

Figure C.10: `invoke-virtual` Semantics Test (Part II)

```
class VirtualTestDriver {
    static public void main( String args[] ) {
        U u = new U();
        U v = new V();
        U w = new W();
        int p = 0;
        VirtualTest tu[] = {
        //          args   result              ame    ire
            new VirtualTest( "",   "U::method()",    false, false ),
            new VirtualTest( "A",  "U::method(A)",   false, false ),
            new VirtualTest( "B",  "U::method(A)",   false, false ),
            new VirtualTest( "C",  "U::method(A)",   false, false ),
            new VirtualTest( "n",  "U::method(A)",   false, false ),
            new VirtualTest( "AA", "U::method(A,A)", false, false ),
            new VirtualTest( "AB", "U::method(A,A)", false, false ),
            new VirtualTest( "AC", "U::method(A,A)", false, false ),
            new VirtualTest( "An", "U::method(A,A)", false, false ),
            new VirtualTest( "BA", "U::method(A,A)", false, false ),
            new VirtualTest( "BB", "U::method(A,A)", false, false ),
            new VirtualTest( "BC", "U::method(A,A)", false, false ),
            new VirtualTest( "Bn", "U::method(A,A)", false, false ),
            new VirtualTest( "CA", "U::method(A,A)", false, false ),
            new VirtualTest( "CB", "U::method(A,A)", false, false ),
            new VirtualTest( "CC", "U::method(A,A)", false, false ),
            new VirtualTest( "Cn", "U::method(A,A)", false, false ),
            new VirtualTest( "nC", "U::method(A,A)", false, false ),
            new VirtualTest( "nB", "U::method(A,A)", false, false ),
            new VirtualTest( "nA", "U::method(A,A)", false, false ),
            new VirtualTest( "nn", "U::method(A,A)", false, false ) };
        for (int i=tu.length-1; i>=0; i--) {
            System.out.print("VirtualTest " + i + " for U ");
            if (false == tu[i].test(u, false)) {
                System.out.println("failed.");
                p++;
            }
        }
        VirtualTest tv[] = {
        //          args   result              ame    ire
            new VirtualTest( "",   "V::method()",    false, false ),
            new VirtualTest( "A",  "U::method(A)",   false, false ),
            new VirtualTest( "B",  "V::method(B)",   false, false ),
            new VirtualTest( "C",  "V::method(B)",   false, false ),
            new VirtualTest( "n",  "",               true,  false ),
            new VirtualTest( "AA", "V::method(A,A)", false, false ),
            new VirtualTest( "AB", "V::method(A,B)", false, false ),
            new VirtualTest( "AC", "V::method(A,B)", false, false ),
            new VirtualTest( "An", "V::method(A,B)", false, false ),
            new VirtualTest( "BA", "",               true,  false ),
            new VirtualTest( "BB", "",               true,  false ),
            new VirtualTest( "BC", "",               true,  false ),
            new VirtualTest( "Bn", "",               true,  false ),
            new VirtualTest( "CA", "",               true,  false ),
            new VirtualTest( "CB", "",               true,  false ),
            new VirtualTest( "CC", "",               true,  false ),
            new VirtualTest( "Cn", "",               true,  false ),
            new VirtualTest( "nC", "",               true,  false ),
            new VirtualTest( "nB", "",               true,  false ),
            new VirtualTest( "nA", "",               true,  false ),
            new VirtualTest( "nn", "",               true,  false ) }
        for (int i=tv.length-1; i>=0; i--) {
            System.out.print("VirtualTest " + i + " for V ");
            if (false == tv[i].test(v, false)) {
                System.out.println("failed.");
                p++;
            }
        }
```

Figure C.11: `invoke-virtual` Semantics Test (Part III)

```
        VirtualTest tw[] = {
        //            args   result           ame    ire
            new VirtualTest( "",   "W::method()",    false, false ),
            new VirtualTest( "A",  "U::method(A)",   false, false ),
            new VirtualTest( "B",  "V::method(B)",   false, false ),
            new VirtualTest( "C",  "",               false, true  ),
            new VirtualTest( "n",  "",               true,  false ),
            new VirtualTest( "AA", "V::method(A,A)", false, false ),
            new VirtualTest( "AB", "V::method(A,B)", false, false ),
            new VirtualTest( "AC", "V::method(A,B)", false, false ),
            new VirtualTest( "An", "V::method(A,B)", false, false ),
            new VirtualTest( "BA", "",               true,  false ),
            new VirtualTest( "BB", "",               true,  false ),
            new VirtualTest( "BC", "W::method(B,C)", false, false ),
            new VirtualTest( "Bn", "W::method(B,C)", false, false ),
            new VirtualTest( "CA", "",               true,  false ),
            new VirtualTest( "CB", "",               true,  false ),
            new VirtualTest( "CC", "W::method(B,C)", false, false ),
            new VirtualTest( "Cn", "W::method(B,C)", false, false ),
            new VirtualTest( "nC", "W::method(B,C)", false, false ),
            new VirtualTest( "nB", "",               true,  false ),
            new VirtualTest( "nA", "",               true,  false ),
            new VirtualTest( "nn", "W::method(B,C)", false, false ) };
        for (int i=tw.length-1; i>=0; i--) {
            System.out.print("VirtualTest " + i + " for W ");
            if (false == tw[i].test(w, false)) {
                System.out.println("failed.");
                p++;
            }
        }
        if (0 == p) System.out.println("Virtual tests passed.");
        else System.out.println("Virtual tests failed = " + p);
    }
}
                    // --->
                    // Virtual tests passed.
```

Figure C.12: `invoke-virtual` Semantics Test (Part IV)

C.2.4 Invoke-static Semantics

```
// StaticTest.java -- test that static multi-dispatch works
// no methods - purely used for argument hierarchy
class A {}
class B extends A {}
class C extends B {}
class S implements StaticMultiDispatchable {
    static String method()           { return "S::method()"; }
    static String method(A a)        { return "S::method(A)"; }
    static String method(B b)        { return "S::method(B)"; }
        static Integer method(C c)   { return new Integer(0); }
    static String method(A a1, A a2) { return "S::method(A,A)"; }
    static String method(B b, A a)   { return "S::method(B,A)"; }
    static String method(A a, B b)   { return "S::method(A,B)"; }
}
class T extends S { // not static multi-dispatchable!
    static String method()    { return "T::method()";    }
    static String method(A a) { return "T::method(A)";   }
}
```

Figure C.13: `invoke-static` Semantics Test (Part I)

```
class StaticTest {
    A args[];
    String result;
    boolean ambiguousmethod;
    boolean illegalreturntype;
    StaticTest(String a, String r, boolean ame, boolean irt) {
        args = new A[a.length()];
        for (int i=0; i<a.length(); i++) {
            switch (a.charAt(i)) {
            case 'A': args[i] = new A(); break;
            case 'B': args[i] = new B(); break;
            case 'C': args[i] = new C(); break;
            case 'n': args[i] = null;    break;
            }
        }
        result = r;
        ambiguousmethod = ame;
        illegalreturntype = irt;
    }
    boolean testS() {
        boolean sa = false;
        boolean si = false;
        String r = "***INVALID***";
        try {
            switch (args.length) {
            case 0: r = S.method(); break;
            case 1: r = S.method(args[0]); break;
            case 2: r = S.method(args[0], args[1]); break;
            }
        } catch (AmbiguousMethodError a) {
            if (false == ambiguousmethod) {
                System.out.println("Error: unexpected AmbiguousMethod thrown");
                return false;
            } else sa = true;
        } catch (IllegalReturnTypeError i) {
            if (false == illegalreturntype) {
                System.out.println("Error: unexpected IllegalReturnType thrown");
                return false;
            } else si = true;
        } catch (Exception e) {
            System.out.println("Error: unexpected " + e + " thrown");
            return false;
        } finally {
            if (sa != ambiguousmethod) {
                System.out.println("Error: expected AmbiguousMethod not thrown");
                return false;
            }
            if (si != illegalreturntype) {
                System.out.println("Error: expected IllegalReturnType not thrown");
                return false;
            }
            if (!sa && !si && r != result) {
                System.out.println("Error: |" + r + "| returned, expected |" + result + "|");
                return false;
            }
        }
        return true;
    }
    boolean testT() { //  identical to testS() but uses class T
    }
}
```

Figure C.14: `invoke-static` Semantics Test (Part II)

```
class StaticTestDriver {
    static public void main( String args[] ) {
        S s = new S();
        S t = new T();
        int p = 0;
        StaticTest ts[] = {
        //          args   result            ame    ire
            new StaticTest( "",   "S::method()",    false, false ),
            new StaticTest( "A",  "S::method(A)",   false, false ),
            new StaticTest( "B",  "S::method(B)",   false, false ),
            new StaticTest( "C",  "",               false, true  ),
            new StaticTest( "n",  "",               false, true  ),
            new StaticTest( "AA", "S::method(A,A)", false, false ),
            new StaticTest( "AB", "S::method(A,B)", false, false ),
            new StaticTest( "AC", "S::method(A,B)", false, false ),
            new StaticTest( "An", "S::method(A,B)", false, false ),
            new StaticTest( "BA", "S::method(B,A)", false, false ),
            new StaticTest( "BB", "",               true,  false ),
            new StaticTest( "BC", "",               true,  false ),
            new StaticTest( "Bn", "",               true,  false ),
            new StaticTest( "CA", "S::method(B,A)", false, false ),
            new StaticTest( "CB", "",               true,  false ),
            new StaticTest( "CC", "",               true,  false ),
            new StaticTest( "Cn", "",               true,  false ),
            new StaticTest( "nA", "S::method(B,A)", false, false ),
            new StaticTest( "nB", "",               true,  false ),
            new StaticTest( "nC", "",               true,  false ),
            new StaticTest( "nn", "",               true,  false ) };
        for (int i=0; i<ts.length; i++) {
            if (false == ts[i].testS()) {
                System.out.println("Test " + i + " for S failed.");
                p++;
            }
        }
        StaticTest tt[] = {
        //          args   result            ame    ire
            new StaticTest( "",   "T::method()",    false, false ),
            new StaticTest( "A",  "T::method(A)",   false, false ),
            new StaticTest( "B",  "T::method(A)",   false, false ),
            new StaticTest( "C",  "T::method(A)",   false, false ),
            new StaticTest( "n",  "T::method(A)",   false, false ) };
        for (int i=0; i<5; i++) {
            if (false == tt[i].testT()) {
                System.out.println("Test " + i + " for T failed.");
                p++;
            }
        }
        if (0 == p) System.out.println("Static tests passed.");
        else System.out.println("Static tests failed = " + p);
    }
}
                    // --->
                    // Static tests passed.
```

Figure C.15: `invoke-static` Semantics Test (Part III)

C.2.5 Invoke-special Semantics

```
// SpecialTest.java --- test that special multi-dispatch works
// no methods - purely used for argument hierarchy
class A {}
class B extends A {}
class U {
  private String method()           { return "U::method()"; }
  private String method(A a)        { return "U::method(A)"; }
  private String method(A a1, A a2) { return "U::method(A,A)"; }
  private String method(B b, A a)   { return "U::method(B,A)"; }
  public void test() {
    A Aa = new A();
    A Ab = new B();
    B Bb = new B();
    try { System.out.println(this.method());           } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa));         } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab));         } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb));         } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null));       } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, Aa));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, Aa));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb, Aa));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null, Aa));   } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, Ab));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, Ab));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb, Ab));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null, Ab));   } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, Bb));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, Bb));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb, Bb));     } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null, Bb));   } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, null));   } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, null));   } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb, null));   } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null, null)); } catch (Error e) { System.out.println("Error: " + e); }
  }
}
```

Figure C.16: `invoke-special` Semantics Test (Part I)

```
class V implements SpecialMultiDispatchable {
  private String method()             { return "V::method()"; }
  private String method(A a)          { return "V::method(A)"; }
  private String method(B b)          { return "V::method(B)"; }
  private String method(A a1, A a2) { return "V::method(A,A)"; }
  private String method(A a, B b)   { return "V::method(A,B)"; }
  private String method(B b, A a)   { return "V::method(B,A)"; }
  public void test() {
    A Aa = new A();
    A Ab = new B();
    B Bb = new B();
    try { System.out.println(this.method());              } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa));            } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab));            } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb));            } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null));          } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, Aa));        } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, Aa));        } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb, Aa));        } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null, Aa));      } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, Ab));        } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, Ab));        } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Bb, Ab));        } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(null, Ab));      } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, Bb));        } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, Bb));        } catch (Error e) { System.out.println("Error: " + e); }
    // these are recognized as ambiguous by the compiler
    // try { System.out.println(this.method(Bb, Bb));     } catch (Error e) { System.out.println("Error: " + e); }
    // try { System.out.println(this.method(null, Bb));   } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Aa, null));      } catch (Error e) { System.out.println("Error: " + e); }
    try { System.out.println(this.method(Ab, null));      } catch (Error e) { System.out.println("Error: " + e); }
    // try { System.out.println(this.method(Bb, null));   } catch (Error e) { System.out.println("Error: " + e); }
    // try { System.out.println(this.method(null, null)); } catch (Error e) { System.out.println("Error: " + e); }
    A anull = null;
    try { System.out.println(this.method(anull, anull));  } catch (Error e) { System.out.println("Error: " + e); }
  }
}
class W {
  W()     { System.out.println("W()");  }
  W(A a) { System.out.println("W(A)"); }
  W(B b) { System.out.println("W(B)"); }
  public static void test() {
    A Aa = new A();
    A Ab = new B();
    B Bb = new B();
    W w = new W();
    w = new W(Aa);
    w = new W(Ab);
    w = new W(Bb);
    w = new W(null);
  }
}
class X implements SpecialMultiDispatchable {
  X()     { System.out.println("X()"); }
  X(A a) { System.out.println("X(A)"); }
  X(B b) { System.out.println("X(B)"); }
  public static void test() {
    A Aa = new A();
    A Ab = new B();
    B Bb = new B();
    X x = new X();
    x = new X(Aa);
    x = new X(Ab);
    x = new X(Bb);
    x = new X(null);
  }
}
```

Figure C.17: `invoke-special` Semantics Test (Part II)

```
class SpecialTestDriver {
    static public void main( String args[] ) {
        U u = new U();
        V v = new V();
        u.test();
        v.test();
        W.test();
        X.test();
    }
}
                    // --->
                    // U::method()
                    // U::method(A)
                    // U::method(A)
                    // U::method(A)
                    // U::method(A)
                    // U::method(A,A)
                    // U::method(A,A)
                    // U::method(B,A)
                    // U::method(B,A)
                    // U::method(A,A)
                    // U::method(A,A)
                    // U::method(B,A)
                    // U::method(B,A)
                    // U::method(A,A)
                    // U::method(A,A)
                    // U::method(B,A)
                    // U::method(B,A)
                    // U::method(A,A)
                    // U::method(A,A)
                    // U::method(B,A)
                    // U::method(B,A)
                    // V::method()
                    // V::method(A)
                    // V::method(B)
                    // V::method(B)
                    // V::method(B)
                    // V::method(A,A)
                    // V::method(B,A)
                    // V::method(B,A)
                    // V::method(B,A)
                    // V::method(A,B)
                    // Error: java.lang.AmbiguousMethodError: in MMDispatch
                    // Error: java.lang.AmbiguousMethodError: in MMDispatch
                    // Error: java.lang.AmbiguousMethodError: in MMDispatch
                    // V::method(A,B)
                    // Error: java.lang.AmbiguousMethodError: in MMDispatch
                    // V::method(A,B)
                    // Error: java.lang.AmbiguousMethodError: in MMDispatch
                    // Error: java.lang.AmbiguousMethodError: in MMDispatch
                    // W()
                    // W(A)
                    // W(A)
                    // W(B)
                    // W(B)
                    // X()
                    // X(A)
                    // X(B)
                    // X(B)
                    // X(B)
```

Figure C.18: `invoke-special` Semantics Test (Part III)

Appendix D

Multi-Dispatch Placement Implementation

D.1 MultiInvoker

```
bool_t
invokeMultiMethod(JHandle *o, struct methodblock *mb, int args_size, ExecEnv *ee) {
    struct methodblock *nmb = NULL;
    int arity;
    switch (*ee->current_frame->lastpc) {
    case opc_invokevirtualobject_quick:
    case opc_invokevirtual:
    case opc_invokevirtual_quick:
    case opc_invokevirtual_quick_w:
    case opc_invokeinterface:
    case opc_invokeinterface_quick:
        nmb = SelectVirtualMultiMethod(o, mb, ee);
        break;
    case opc_invokestatic:
    case opc_invokestatic_quick:
        nmb = SelectStaticMultiMethod(mb, ee);
        break;
    case opc_invokesuper_quick:
        nmb = SelectSuperMultiMethod(mb, ee);
        break;
    case opc_invokespecial:
        if (IsPrivate(mb)) nmb = SelectSpecialMultiMethod(mb, ee);
        else if (fieldname(&mb->fb) == utf8_literal_init_obj_name) nmb = SelectSpecialMultiMethod(mb, ee);
        else nmb = SelectSuperMultiMethod(mb, ee);
        break;
    case opc_invokenonvirtual_quick:
        nmb = SelectSpecialMultiMethod(mb, ee);
        break;
    }
    if (NULL == nmb) { /* Select*MultiMethod already threw the error */
        return FALSE;
    }
    if ((nmb == mb) || (fieldsig(&mb->fb) == fieldsig(&nmb->fb)))
        return (methodCachedInvoker(nmb))(o, nmb, nmb->args_size, ee);
    if (VerifyReturnType(mb, nmb, ee)) return (methodCachedInvoker(nmb))(o, nmb, nmb->args_size, ee);
    else { /* IllegalReturnTypeError already thrown */
        return FALSE;
    }
}
```

Figure D.1: MultiInvoker Implementation

D.2 Inlined Multi-Dispatch

```
;************* MULTIMETHODS *************
; macro that expands into inline test for ACC_MULTIMETHOD in mb->fb.access
; and branches to specialized code multimethod flag set
;
; parameters: $1 = VIRTUAL, STATIC, SPECIAL
;             $2 = label to contine at if not mmd method
;             $3 = 32-bit register that can be trashed
;
; registers:  eax = uni-mb (unchanged)
;             $3 and condition register trashed
;
define('CHECKMULTI','
    ifdef('MMD_'$1,'
  movzx $3, WORD PTR [eax + mb_fb + fb_access]   ; mb->fb.access
  and $3, FJ_ACC_'$1'_MMD                        ; set condition register
  ;;; debug macro removed for clarity
  jnz Lb('$2'_call_multi_method)                 ; multimethod!
        ':,'')')
;************* MULTIMETHODS *************
```

Figure D.2: Multi-Dispatch Assembler Interpreter Loop — Inline Test Macro

Appendix E

Multi-Dispatcher Implementation

E.1 MSA Algorithm

```
/* SelectVirtualMultiMethod() -- select a virtual multi-method */
struct methodblock *SelectVirtualMultiMethod(const JHandle *o, struct methodblock *mb, ExecEnv *ee) {
    ClassClass *receiverclass, *hostclass;
    struct methodblock **nextRead, **lastRead, **nextWrite;
    struct methodblock *method, *lastmethod, *bestMethod;
    struct methodblock *candidates[MMD_MAXIMUM_CANDIDATES];
    const char *methodName = fieldname(&mb->fb);
    const int argCount = methodArity(mb);
    receiverclass = obj_array_classblock(o);
    if (NULL == ee->current_frame->current_method) hostclass = NULL;
    else hostclass = fieldclass(&(ee->current_frame->current_method)->fb);
    for (lastRead = cbMethodTable(receiverclass)->methods,
             nextRead = cbMethodTable(receiverclass)->methods + cbMethodTableSize(receiverclass) - 1,
             nextWrite  = candidates;
         nextRead > lastRead;
         nextRead--) {
        if ((mb == *nextRead)
            || ((methodName == fieldname(&(*nextRead)->fb))
                && (argCount == methodArity(*nextRead))
                && IsVirtual(*nextRead)
                && IsMultiMethod(*nextRead)))
            *nextWrite++ = *nextRead;
        }
    }
    /* add any private methods defined in the receiver class if appropriate */
    for (method = cbMethods(receiverclass),     /* continuous block of methods */
             lastmethod = cbMethods(receiverclass) + cbMethodsCount(receiverclass);
         method < lastmethod; /* 0-based array indexing */
         method++) {
        if ((methodName == fieldname(&method->fb))
            && (argCount == methodArity(method))
            && !IsStatic(method)
            && IsPrivate(method->fb.access)
            && IsMultiMethod(method)
            && canAccess(hostclass, method)) {
            *nextWrite++ = method;
    }
    bestMethod = selectMostSpecificMethod(mb, candidates, nextWrite, ee);
    if ((NULL != bestMethod) && VerifyReturnType(mb, bestMethod, ee)) return bestMethod;
    else return NULL;
}
```

Figure E.1: `SelectVirtualMultiMethod` for MSA Technique

```
/* SelectStaticMultiMethod() -- select a static multi-method */
struct methodblock *SelectStaticMultiMethod(struct methodblock *mb, ExecEnv *ee) {
    ClassClass *class, *hostclass;
    ClassClass *argTypes[MMD_MAX_ARITY];
    uint32_t nullArgs, nonNullArgs;
    struct methodblock *method, *lastmethod;
    struct methodblock **nextRead, **lastRead, **nextWrite;
    struct methodblock *bestMethod, *tentative;
    struct methodblock *candidates[MMD_MAXIMUM_CANDIDATES];
    const char *methodName = fieldname(&mb->fb);
    const int argCount = methodArity(mb);
    class = fieldclass(&mb->fb);
    if (NULL == ee->current_frame->current_method) hostclass = NULL;
    else hostclass = fieldclass(&(ee->current_frame->current_method)->fb);
    for (nextWrite = candidates,
              method = cbMethods(class),
              lastmethod = cbMethods(class) + cbMethodsCount(class);
         method < lastmethod;
         method++) {
        if ((methodName == fieldname(&method->fb))
            && (argCount == methodArity(method))
            && IsStatic(method)
            && IsMultiMethod(method))
            *nextWrite++ = method;
    }
    bestMethod = selectMostSpecificMethod(mb, candidates, nextWrite, ee);
    if ((NULL != bestMethod) && VerifyReturnType(mb, bestMethod, ee)) return bestMethod;
     else return NULL;
}
```

Figure E.2: `SelectStaticMultiMethod` for MSA Technique

```
/* SelectSpecialMultiMethod() -- select a multi-method for an invoke-special (not super) */
struct methodblock *SelectSpecialMultiMethod(struct methodblock *mb, ExecEnv *ee) {
    ClassClass *hostclass;
    ClassClass *argTypes[MMD_MAX_ARITY];
    uint32_t nullArgs,  nonNullArgs;
    struct methodblock *method, *lastmethod;
    struct methodblock **nextRead, **lastRead,  **nextWrite;
    struct methodblock *bestMethod,  *tentative;
    struct methodblock *candidates[MMD_MAXIMUM_CANDIDATES];
    const char *methodName = fieldname(&mb->fb);
    const int argCount = methodArity(mb);
    hostclass = fieldclass(&mb->fb);
    for (method = cbMethods(hostclass),
              lastmethod = cbMethods(hostclass) + cbMethodsCount(hostclass),
              nextWrite  = candidates;
         method < lastmethod;
         method++) {
        if ((methodName == fieldname(&method->fb))
            && (argCount == methodArity(method))
            && IsVirtual(method)
            && IsMultiMethod(method))
            *nextWrite++ = method;
    }

    bestMethod = selectMostSpecificMethod(mb, candidates, nextWrite, ee);
    if ((NULL != bestMethod) && VerifyReturnType(mb, bestMethod, ee)) return bestMethod;
    else return NULL;
}
```

Figure E.3: `SelectSpecialMultiMethod` for MSA Technique

```
/* SelectSuperMultiMethod() -- select a multi-method for invokespecial (super only) */
struct methodblock *SelectSuperMultiMethod(struct methodblock *mb, ExecEnv *ee) {
    ClassClass *sclass, *hostclass;
    struct methodblock **nextRead, **lastRead, **nextWrite;
    struct methodblock *bestMethod;
    struct methodblock *candidates[MMD_MAXIMUM_CANDIDATES];
    const char *methodName = fieldname(&mb->fb);
    const int argCount = methodArity(mb);
    sysAssert(NULL != ee->current_frame->current_method);
    hostclass = fieldclass(&(ee->current_frame->current_method)->fb);
    sclass = cbSuperclass(hostclass);
    for (lastRead = cbMethodTable(sclass)->methods,
            nextRead = cbMethodTable(sclass)->methods + cbMethodTableSize(sclass) - 1,
            nextWrite  = candidates;
         nextRead > lastRead;
         nextRead--) {
        if ((mb == *nextRead)
            || ((methodName == fieldname(&(*nextRead)->fb))
                && (argCount == methodArity(*nextRead))
                && IsVirtual(*nextRead)
                && IsMultiMethod(*nextRead)))
            *nextWrite++ = *nextRead;
        }
    }
    bestMethod = selectMostSpecificMethod(mb, candidates, nextWrite, ee);
    if ((NULL != bestMethod) && VerifyReturnType(mb, bestMethod, ee)) return bestMethod;
    else return NULL;
}
```

Figure E.4: `SelectSuperMultiMethod` for MSA Technique

```
static struct methodblock *
    selectMostSpecificMethod(struct methodblock *mb, struct methodblock **candidates,
                             struct methodblock **lastCandidate, ExecEnv *ee) {
    uint32_t nullArgs, nonNullArgs;
    ClassClass *argTypes[MMD_MAX_ARITY];
    struct methodblock *tentative;
    struct methodblock **nextCandidate, **nextWrite;
    if (candidates == (lastCandidate - 1)) return (struct methodblock *) mb;
    if (!getArgumentTypes(mb, ee->current_frame->optop, argTypes, &nullArgs)) {
        ThrowInternalError(0, "getting arguments from stack");
        return NULL;
    }
    nonNullArgs =  nullArgs;
    tentative = NULL;
    for (nextCandidate = candidates; nextCandidate < lastCandidate; nextCandidate++) {
        if (methodParamsToResolve(*nextCandidate) && (methodParamsToResolve(*nextCandidate) & nonNullArgs)) {
            if (!ResolveParamTypes(*nextCandidate, ALL_RESOLVED, ee)) return NULL;
            if (methodParamsToResolve(*nextCandidate) & nonNullArgs) continue;
        }
        if ((mb == *nextCandidate) || (isApplicable(ee, *nextCandidate, argTypes, nullArgs))) {
            tentative = *nextCandidate;
            break;
        }
    }
    nextCandidate++;
    for (nextWrite = candidates; nextCandidate < lastCandidate; nextCandidate++) {
        if (methodParamsToResolve(*nextCandidate) && (methodParamsToResolve(*nextCandidate) & nonNullArgs)) {
            if (!ResolveParamTypes(*nextCandidate, ALL_RESOLVED, ee)) return NULL;
            if (methodParamsToResolve(*nextCandidate) & nonNullArgs) continue;
        }
        if ((mb == *nextCandidate)
            || isApplicable(ee, *nextCandidate, argTypes, nullArgs))
            if (isMoreSpecificMethod(ee, *nextCandidate, tentative, nullArgs))
                if (nullArgs) *nextWrite++ = *nextCandidate;
                else tentative = *nextCandidate;
            else if (!isMoreSpecificMethod(ee, tentative, *nextCandidate, !nullArgs))
                *nextWrite++ = *nextCandidate;
    }
    if (nextWrite == candidates) return tentative;
    if (nullArgs) {
        *nextWrite++ = candidates[0];
        candidates[0] = tentative;
        tentative = NULL;
        for (nextCandidate = candidates, lastCandidate = nextWrite; nextCandidate < lastCandidate; nextCandidate++) {
            if (methodParamsToResolve(*nextCandidate)
                && (methodParamsToResolve(*nextCandidate) & nullArgs)
                && (!ResolveParamTypes(*nextCandidate, nullArgs, ee))) return NULL;
            if ((mb == *nextCandidate) || isApplicableOnly(ee, *nextCandidate, argTypes, FALSE, nullArgs)) {
                tentative = *nextCandidate;
                nextCandidate++;
                break;
            }
        }
        for (nextWrite = candidates; nextCandidate < lastCandidate; nextCandidate++) {
            if (methodParamsToResolve(*nextCandidate)
                && (methodParamsToResolve(*nextCandidate) & nullArgs)
                && (!ResolveParamTypes(*nextCandidate, nullArgs, ee))) return NULL;
            if ((mb == *nextCandidate) || isApplicableOnly(ee, *nextCandidate, argTypes, FALSE, nullArgs)) {
                if (isMoreSpecificMethod(ee, *nextCandidate, tentative, FALSE)) tentative = *nextCandidate;
                else if (!isMoreSpecificMethod(ee, tentative, *nextCandidate, TRUE)) *nextWrite++ = *nextCandidate;
            }
        }
    }
    if (nextWrite == candidates) return tentative;
    for (nextCandidate = candidates, lastCandidate = nextWrite, nextWrite = candidates;
         nextCandidate < lastCandidate;
         nextCandidate++)
        if (!isMoreSpecificMethod(ee, tentative, *nextCandidate, TRUE)) *nextWrite++ = *nextCandidate;
    if (candidates != nextWrite) {
        *nextWrite++ = tentative;
        ThrowAmbiguousMethodError(ee, fieldname(&mb->fb), candidates, nextWrite);
        return NULL;
    }
    return tentative;
}
```

Figure E.5: Inner Dispatcher for MSA Technique

E.2 Tuned SRP Algorithm

```
/* some shorthand to make things more legible */
typedef struct methodblock *Method;
typedef ClassClass         *Class;
struct Behaviour {
    short int nBits;             /* 0, 32, 64, 128, 256 == allocated space for implementations */
    short int nDispatchedSlots;  /* # rows in bits array == # bits set in dispatchedSlots */
    u32       dispatchedSlots;   /* bitfield in rev order of dispatched slots ** D, L take 2 slots */
    void     **bits;             /* uXX bits[nDispatchedSlots][numTypeNums] */
    Method    *impls;            /* method implementations */
    void      *unresolved;       /* uXX bitfield of unresolved methods */
    void      *toresolve;        /* uXX bitfield of methods that need resolution */
    void      *overrides;        /* uXX[nImpls] bitfield of methods overrridden */
    void      *implsbits;        /* uXX bitfield each method implementation registered */
  /* put these last for fastest dispatch */
    Class       type;            /* originating class - models behaviour inheritance */
    char       *selector;        /* method name */
    char       *sig;             /* terse signature without return type */
    short int   arity;           /* this is args not slots */
    short int   nImpls;          /* number of implementations used (always <= nBits) */
}
```

Figure E.6: SRP Dispatcher — Behaviour Structure

```
/*
 * MMDispatch() -- the heart of the whole thing
 */
INLINE Method MMDispatch(Method mb, ExecEnv *ee) {
    TypeNum tnums[MMD_MAX_ARITY];
    Behaviour b = methodBehaviour(mb);
    Method mmb;
    sys_thread_t *self = sysThreadSelf();
    DISPATCH_LOCK(b, self);
    getArgumentTypeNums(b, ee->current_frame->optop, tnums);
    switch (b->nBits) {
    case 32:    mmb = MMDispatch32(b, tnums);   break;
    /* 64, 128, 256 bit versions similar */     }
    DISPATCH_UNLOCK(b, self);
    if (NULL != mmb) {
        if (VerifyReturnType(mb, mmb, ee)) return mmb;
        else return NULL;
    else if (exceptionOccurred(ee)) return NULL;
    ResolveForDispatch(b);
    return MMDispatch(mb, ee);
}


/*
 * MMSuperDispatch() -- the heart of the whole thing
 */
INLINE Method MMSuperDispatch(Method mb, TypeNum super, ExecEnv *ee) {
    TypeNum tnums[MMD_MAX_ARITY];
    Behaviour b = methodBehaviour(mb);
    Method mmb;
    sys_thread_t *self = sysThreadSelf();
    DISPATCH_LOCK(b, self);
    getArgumentTypeNums(b, ee->current_frame->optop, tnums);
    tnums[0] = super;
    switch (b->nBits) {
    case 32:    mmb = MMDispatch32(b, tnums);   break;
    /* 64, 128, 256 bit versions similar */     }
    DISPATCH_UNLOCK(b, self);
    if (NULL != mmb)
        if (VerifyReturnType(mb, mmb, ee)) return mmb;
        else return NULL;
    else if (exceptionOccurred(ee)) return NULL;
    ResolveForDispatch(b);
    return MMSuperDispatch(mb, super, ee);
}
```

Figure E.7: `MMDispatch` — The Outer Dispatchers

```
/* the actual dispatch algorithm */
INLINE static Method MMDispatch32(Behaviour b, TypeNum *tnums) {
    int r, o;
    int nr = b->nDispatchedSlots;
    u32 mbits;
    copy32(mbits, *((u32*) b->implsbits));
    for (r=0; r<nr; r++)
        if (TYPE_NULL < tnums[r]) and32(mbits, ((u32 **) b->bits)[r][tnums[r]]);

    /* we have the bitfield of methods applicable to this call site */
    o = ffs32(mbits);
    if (0 > o) { /* no methods apply! */
        ThrowNoSuchMethodError(EE(), "in MMDispatch");
        return NULL;
    }
    clear32(mbits, o);
    if (isZero32(mbits)) /* only one set ... */
        return b->impls[o];
    }
    /* eliminate methods overridden by the most specific applicable */
    and32(mbits, ((u32*) b->overrides)[o]);
    if (isZero32(mbits)) return b->impls[o];
    copy32(*((u32 *)b->toresolve), mbits);
    set32(*((u32 *)b->toresolve), o);
    and32(*((u32 *)b->toresolve), *((u32 *) b->unresolved));
    if (isNotZero32(*((u32 *)b->toresolve))) /* NULL w/o exception -> additional resolution */
        return NULL;
    {   /* ambiguity */
        int i, n;
        Method *methods;
        Method *lmethod;
        for (i=0, n=1; i<32; i++)
            if (isSet32(mbits, i)) n++;
        methods = (Method *) sysCalloc(n, sizeof(Method));
        if (NULL == methods) {
            ThrowOutOfMemoryError(EE(), "in MMDispatch");
            return NULL;
        }
        lmethod = &methods[n];
        methods[0] = b->impls[o];
        for(i=0, n=1; i<32; i++)
            if (isSet32(mbits, i)) methods[n++] = b->impls[i];
        ThrowAmbiguousMethodError(EE(), "in MMDispatch", methods, lmethod);
    }
    return NULL;
}
```

Figure E.8: 32-Bit Implementation of Inner Dispatcher

```
/* can only resolve them one at a time, because resolution *WILL* load
   more classes, which may load more implementations which may change
   the shape of the behaviour. */
static void ResolveForDispatch(Behaviour b) {
    int o = -1;
    Method mb;
    switch (b->nBits) {
    case 32: o = ffs32(*((u32 *)b->toresolve)); break;
    /* 64, 128, 25 are similar */
    }
    if (0 > o) return;
    mb = b->impls[o];
    ResolveParamTypeNums(mb, NONE_RESOLVED, EE());
    if (exceptionOccurred(EE())) { return; }
    moveImpl(b, mb);
    ResolveForDispatch(b);
}
```

Figure E.9: Resolver

```
/*
 * SelectVirtualMultiMethod() -- select a virtual multi-method
 */
Method SelectVirtualMultiMethod(const JHandle *o, Method mb, ExecEnv *ee) {
    Method mmb = MMDispatch(mb, ee);
    if (!canAccess(obj_array_classblock(EE()->current_frame->optop->h->obj), mmb)) {
        ThrowIllegalAccessError(EE(), "in SVMM");
    }
    return mmb;
}
/*
 * SelectStaticMultiMethod() -- select a static multi-method
 */
Method SelectStaticMultiMethod(Method mb, ExecEnv *ee) {
    Method mmb = MMDispatch(mb, ee);
    if (!canAccess(obj_array_classblock(EE()->current_frame->optop->h->obj), mmb)) {
        ThrowIllegalAccessError(EE(), "in SVMM");
    }
    return mmb;
}
/*
 * SelectSpecialMultiMethod() -- select a multi-method for an invoke-special (not super)
 */
Method SelectSpecialMultiMethod(Method mb, ExecEnv *ee) {
    Method mmb = MMDispatch(mb, ee);
    /* no need to check access - this method is certainly self-contained */
    return mmb;
}
/*
 * SelectSuperMultiMethod() -- select a multi-method for invokespecial (super only)
 */
Method SelectSuperMultiMethod(Method mb, ExecEnv *ee) {
    Class host = fieldclass(&(ee->current_frame->current_method->fb));
    TypeNum super;
    Method mmb;
    sysAssert(TYPE_NULL < cbTypeNum(host));
    sysAssert(NULL != cbSuperclass(host));
    super = cbTypeNum(cbSuperclass(host));
    sysAssert(TYPE_NULL < super);
    /* need to ensure we use the correct the first argument type */
    mmb = MMSuperDispatch(mb, super, ee);
    if (!canAccess(obj_array_classblock(EE()->current_frame->optop->h->obj), mmb)) {
        ThrowIllegalAccessError(EE(), "in SVMM");
    }
    return mmb;
}
```

Figure E.10: `Select-*-MultiMethod` Routines